Lean Learning - Applying Lean Techniques to Improve Software Engineering Education

Robert Chatley
Imperial College London
180 Queen's Gate
London, UK
rbc@imperial.ac.uk

Tony Field
Imperial College London
180 Queen's Gate
London, UK
ajf@imperial.ac.uk

Abstract—Building a programme of education that reflects and keeps pace with industrial practice is difficult. We often hear of a skills shortage in the software industry, and the gap between what people are taught in university and the "real world". This paper is a case study showing how we have developed a programme at Imperial College London that bridges this gap, providing students with relevant skills for industrial software engineering careers. We give details of the structure and evolution of the programme, which is centred on the tools, techniques and issues that feature in the everyday life of a professional 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. The contributions of this paper take the form of lessons learnt, which may be seen as recommendations for others looking to evolve their own teaching structures and methods.

Keywords-education; software engineering; curriculum design;

I. INTRODUCTION

In order to provide training in the types of software engineering methods and practices that are used in industrial development projects, many universities and other higher-education institutions are striving to bring modern industrial software development techniques into the classroom. Keeping pace with rapid changes in industrial practice has required changes in the way software engineering is taught. Like any such institution, Imperial College has been faced with the challenge of updating and evolving its software engineering education to prepare its students for modern industrial careers. This includes teaching modern development methods and giving students hands-on experience of putting those methods into action through practical work [1], [2]. 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.

One of the main challenges we have faced is to teach software engineering in a way that allows us to give sustained, high-quality feedback and guidance to all of our students, even in the face of large class sizes. In our case we have classes of up to 150 students; class sizes at some other institutions are larger still. A key lesson we have learnt from iterating on our course formats is that to achieve high quality education, we must aim for *lean learning*, with fast feedback and short cycle times, working in small frequent increments. In other words, we argue that the learning experiences that we provide for our students should be aligned with the principles promoted by the agile methods that we are trying to teach, and which industry has adopted widely for the production of quality software [3]. Properly administered, the application of lean principles to content delivery and to assessment reduces the burden on both students and instructors, and we believe that at the same time this can enhance the learning experience.

The objective of this paper is to describe our experiences, both successes and failures, documenting what we have learnt. Although Imperial's student intake, learning culture and degree structures may be very different from other institutions, we believe that the principles we have adopted are widely applicable and that they scale well to large classes for which, invariably, there is limited teaching support.

The contributions of this paper are in the form of the lessons learnt, which may be seen as recommendations for others looking to evolve their own teaching structures and methods. In summary these are:

- Students should learn by doing and, wherever possible, software engineering principles should be assessed in the context of practical work, rather than by regurgitating material taught or extracted from text books.
- Assessment should be done "little and often". Frequent, small assignments align perfectly with the principles of modern agile development and facilitate sustained high-quality feedback. We believe that setting many small goals avoids large peaks of exertion and stress for both students and staff, and that this allows students to learn more than they might otherwise, at a sustainable pace.
- Good tools reduce the assessment burden and instil good practice in the students' workflows. Time spent on tooling and automation is therefore time well spent.
- Industry engagement in the teaching of software engineering is essential. Instructors and guest contributors who are themselves practising software engineers are often much better placed to teach practical software engineering skills than pure academics, and naturally lend a greater sense of relevance to the material being taught.

We revisit these points in more detail in the discussion section at the end of the paper (Section VII).

The remainder of this paper is structured around four key components of software engineering education that have guided the development of our own teaching programmes. The first concerns the set of core skills that we expect every student to acquire prior to learning wider aspects of software engineering, which includes the ability to program "in the small" (Section III). The second concerns the transition between programming "in the small" and developing, testing and deploying large software systems; we refer to this as as software engineering *design* (Section IV). The third is team project work, including project management (Section V), which is about working together to build a product or service that fulfils the needs of particular users. The fourth is industrial practice and experience (Section VI), which is about providing students with the tools to appreciate, explore and debate issues facing industrial software engineers in their daily work. It so happens that Imperial's curriculum has one major module associated with each of the last three aspects described, but the focus of the paper is the curriculum content and delivery methods, not the structure of individual modules.

II. SOFTWARE ENGINEERING EDUCATION

The majority of students entering a specialist computer science or software engineering programming are looking to gain industry-relevant skills and knowledge to further their future careers in the growing technology sector. Our learning objectives in designing a programme must then be aligned to and informed by the needs of industry. The software engineering content of our degree programmes is centred on the four components outlined above (core skills, design, group work and industrial practice). In the following sections we detail an approach for teaching these topics, and most importantly, allowing students to practise skills, and to obtain feedback on their progress.

It should be noted that there are many other aspects of the computer science curriculum that one could consider to fall under the umbrella of software engineering, but that we do not address here. Whilst our own curriculum has many courses addressing topics like formal methods (formal specification, verification, model checking, etc.), software security, performance engineering and so on, this paper focuses on the more practical aspects of software design and delivery, as broadly exemplified by the first eight elements of the "SE. Software Engineering" component of the ACM Computer Science Curricula (2013) [4].

A. Industrial Landscape

Together with more formal surveys [3], a quick search online for software development job adverts will demonstrate that the vast majority of software development teams working today are employing some kind of agile method (or at least proclaiming to do so). The aim of these methods is to improve efficiency and effectiveness in teams delivering software.

There are several popular development methods or processes that come under the banner of agile software development. Extreme Programming (XP) [5] is one of the original agile methods, and includes project management techniques as well as technical practices to help to deliver reliable software quickly. Scrum [6] concentrates more on the project management methods, and does not talk specifically about building software. Both XP and Scrum focus on delivering software iteratively and incrementally in fixed length cycles - typically timeboxes of between one and three weeks.

Kanban [7] is a more recent method influenced by Japanese manufacturing techniques, particularly from companies like Toyota, that aims for a continuous flow of work. It is based on the principles of lean manufacturing [8] that focuses on eliminating waste to increase throughput, but does not work in a regular iteration cycle.

In recent years, *continuous delivery* [9] has become popular and widely adopted, where every individual change to a piece of software should produce a potentially shippable product increment. Through the use of automation, the batch size of changes can be made small, reducing lead time to delivery without compromising engineering rigour or quality.

B. Staffing

Classes in research-led universities are almost always taught by academics, but few academics have personal experience of developing software in an industrial environment. While many academics, particularly computer scientists, do write software as part of their research work, the way in which these development projects are carried out is normally not representative of the way that projects are run in industrial settings. Researchers predominantly work on fairly small software projects that act as prototypes or proofs-of-concept to demonstrate research ideas. As such they do not have the pressures of developing robust software to address a mass market. They may concentrate on adding new features required to further their research, paying less attention to robustness or maintainability. They do not typically have a large population of users to support, or need support the operation of a system that runs 24/7, as the developers of an online retailer, financial services organisation or telecoms company might.

Furthermore, even within large research groups, academics and postgraduate researchers often work on their own, and so often do not have experience of planning and managing the work of many different contributors to a software project, integrating all of these whilst preserving an overall architecture that supports maintainability, and making regular releases to a customer according to an agreed schedule. Because of this, few academics have occasion to develop practical experience of the project management and quality assurance methods prevalent in modern industrial software development.

Our approach to tackling these issues is, wherever possible, to engage members of the industrial software engineering community to aid in teaching our software engineering curriculum, drawing on their practical experience to guide content and the delivery. This has ranged from offering advice on specific

current issues, helping to outline course content, delivering guest lectures or coaching and, to greatest effect, joining the faculty staff. One of the senior teaching fellows at Imperial is employed on a part-time basis to teach much of the software engineering content, whilst also working as a practising software engineer. This has been instrumental in aligning our material with current industrial practices as well as providing access to external contributors through professional contacts. We have found that practitioners are generally very happy to help shape the curriculum for the next generation of software engineers and to give something back to the community. They can also benefit by exposing students to their own areas of business and this in turn benefits both students and companies when it comes to recruitment.

C. Perspectives on Teaching

Over the past five years, we have refined our courses in a series of iterations. In order to discuss the approaches that we have tried, we borrow some vocabulary from [10]. This 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 is *apprenticeship*, which refers to 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 practise a recipe repeatedly until they have mastered a dish. The third 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 universities typically do not have the resources 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.

D. Brief Overview of the Imperial Programme

Like many UK institutions, Imperial runs three-year Bachelors and four-year "integrated" Masters degrees in Computing at undergraduate level. The first three years are fundamentally the same across both programmes, but those going on to take the fourth year also undertake a six month industrial work placement between their (shortened) third and fourth years.

Each year is divided into three *terms*, with the bulk of the classroom teaching happening in the Autumn (Fall) and Spring terms which are 11-weeks long and run either side of a Christmas break. The third, Summer, term is separated from the Spring term by an Easter break; it is slightly shorter than the other two terms and is primarily reserved for project work.

At the end of each term students complete an anonymous on-line survey. This feedback has been instrumental in guiding the evolution of our programme and, where appropriate, we include selected comments in the paper, both positive and negative, as part of the narrative.

III. TEACHING CORE SKILLS

It seems futile to teach software engineering before learning to program, so the first year of Imperial's degree places a great deal of emphasis on programming "in the small". Students are taught to program in a variety of paradigms (functional, object-oriented, procedural), are exposed to several languages (presently Haskell, Java, C and assembler) and are expected to be able to produce succinct, elegant solutions to reasonably well-specified problems using appropriate language features and abstractions, including those for basic concurrency and parallelism. We believe that high quality feedback and high-touch personalised support is crucial during these early stages. To achieve this, core programming skills are acquired primarily through the completion of weekly exercises. These are assessed and fed back in small-group tutorials led by senior undergraduate students acting as Undergraduate Teaching Assistants (UTAs) working alongside faculty staff [11]. To stress the importance of good tools, all students are required to use Git[1] for version control from day one. To add to the feedback from the weekly tutorials, students must pass a series of practical on-line programming tests throughout the year in order to progress to subsequent years.

Crucially, in addition to learning programming, students are exposed to a wide spectrum of computer science topics in the first two years. This includes the structure and operation of computer systems (computer hardware, architecture, networking, compilers, databases and operating systems), formal reasoning about programs and systems (logic, discrete structures, program reasoning, computational models, algorithms and complexity) and the continuous mathematics that underpins both analysis and some of the key applications of computing. By the time students are introduced to software engineering, at the start of the second year, we know that they are proficient programmers with an appreciation of the core computer science principles that guide and influence the design of software.

IV. SOFTWARE ENGINEERING DESIGN

Within Imperial's curriculum, software engineering design concerns the 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*. This is taught within a single course of the same name. In an earlier iteration of the course, the material concentrated on notation, formal specification languages and catalogues of design patterns [12]. This meant that students would know

---

[1]https://git-scm.com/

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 learning outcomes did not align well with mainstream industrial practice. The following survey comment typified these concerns:

> "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..."

It is worth spending a moment to consider the issue of formal methods when teaching software engineering. Formal specification techniques are used by engineers developing safety-critical and high-precision systems, but these make up only a small proportion of industrial teams. Many more are working on doubtless important - but not safety-critical - systems that support business in different types of enterprise, consumer web services, apps and games etc. The use of formal specification techniques among such teams is relatively rare. Also, as agile development methods are now prevalent [3], design is no longer considered a separate phase of the project, to be completed before coding commences - rather it is a continuous process of decision making and change as the software evolves over many iterations. There are still design concerns at play, but rather than needing a way to specify a software design abstractly up-front, the common case is that team members need ways to discuss and evaluate design ideas when considering how to make changes and add new features to an existing piece of software.

Our philosophy is thus to think of design as a constant effort to maintain a codebase in a way that makes it amenable to continuous evolution and change [13].

A. Batch Sizing for Feedback

Historically, teaching in our software design course was based largely on transmission, referring back to the terminology of [10]. Students attended lectures twice per week throughout the autumn, took other modules during the spring, and then had their examinations after Easter, as shown in Fig. 1. There were tutorial classes alongside the lectures,
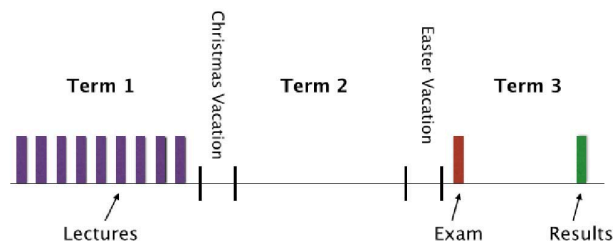

Fig. 1. Traditional lecture course schedule

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.
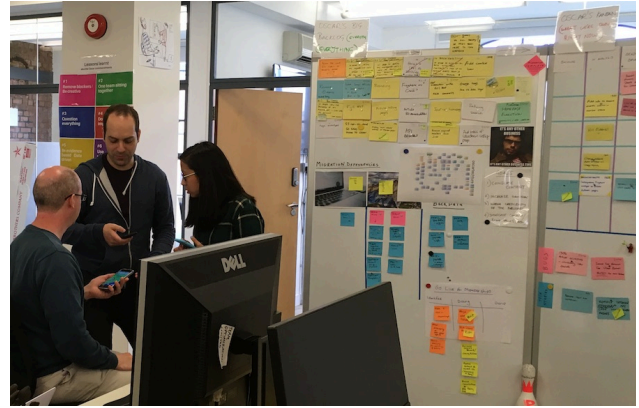

Fig. 2. This agile team organises their work using a physical Kanban board.

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's 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 methods. Kanban is a lean method that focusses on flow through a system, and by using it we can aim to maximise throughput and minimise cycle time. 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.

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 (see Fig. 2). The different columns map the "value stream" [14]. 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. 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 [15]. 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
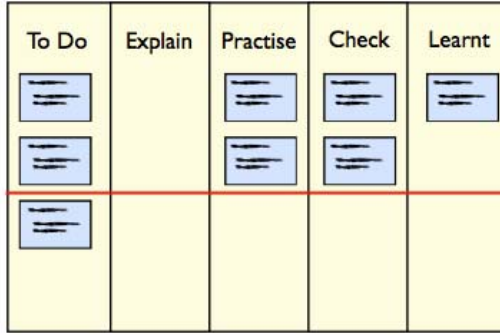
Fig. 3.   Value stream of learning

syllabus as our backlog - "to do" - and then have columns for "explain" (transmission), "practise" (apprenticeship), "check" and "learnt" (fig 3). 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 practise 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.

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 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 constructing a number of small scenarios, each student had the same design issues to think about, and by making them into coding examples students get a much more hands-on, kinaesthetic learning experience.

This "lean" approach leads to a weekly cycle of assessment, as shown in Fig. 4. A new topic is addressed each week with an associated assignment, and students submit their solution by the same time the following week. Grades and feedback are then returned within 4 or 5 working days, i.e. before they submit their next assignment.

B. Achieving Weekly Feedback

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 two weeks, in order to be able 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. By adopting a process of *continuous integration* [16] 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" [17].

A key to reducing the burden of assessment and feedback is to add automation. Our approach here has been to provide tools that enable students to test their solutions as they work. From the first week of their first year students learn to use version control through Git and GitLab[2]. When they start an 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 Lab Test System (LabTS), which allows students to view and test each version of their code.

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, when learning about software design, we do not want students to follow the same approach. Providing a test suite has a consequence of defining an API that the students need to implement. Here 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 for that week's exercise. LabTS gives each submission a score out of 5: 1 point if the code compiles, 1 point if it passes some basic linting and formatting checks, 1 point if all the tests the students have written pass, and 2 points if these tests meet the code coverage threshold. The exercises for our second-year design course are in Java, so we use a Maven[3] build to choreograph the compilation, testing and other checks. We configure Maven plugins to check code formatting against a given style guide, and to measure test coverage. LabTS is then set up to run this

[2]https://about.gitlab.com/
[3]https://maven.apache.org/

Maven build against each submission and report the results. Usually if a LabTS test run does not score 5/5, 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 5/5 on LabTS then a human marker does not need to look at it. Once they have 5/5 on LabTS then a human marker can give more nuanced feedback on the design, and should not have to pick up on basic points about compilation, style, or test coverage. 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.

One further change that made a big difference to both student learning and marking load is to encourage students to pair-program, which has been shown to be highly effective in a classroom environment [18]. 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 practise 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. Lastly, a major benefit in terms of giving weekly feedback on assignments was that pair-programming reduced the number of submissions from 150 to 75!

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 two hours each week.

We have found that this 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 practise and to improve. 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."
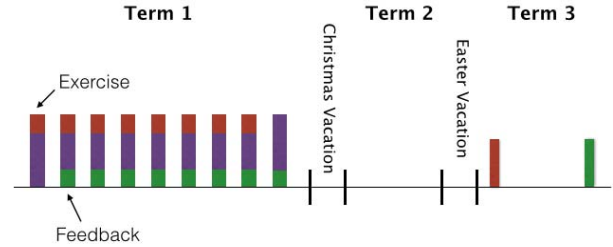

Fig. 4. Weekly exercises and weekly feedback. Red shows when assessments (exercises or exams) are set, and green when feedback is returned. As before, purple shows transmission of content through lectures.

V. SOFTWARE ENGINEERING GROUP 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. At Imperial, students get experience of working in small groups from as early as the first year, but the third year *group projects* represent their main exposure to teamwork on a larger scale. Students form groups of 5 or 6 and are given a major assignment to work on over a period of about 3 months (in parallel with other taught courses). The projects are assigned at the start of Term 1 with the project being demonstrated and the final assessment happening immediately after the Christmas break (Fig. 5).

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 [19], systems for estimating heart rate based on video or speech recordings, and verifying product provenance using BlockChain technology. Each group has a supervisor – a member of the faculty, or an industrial partner – who acts as a customer to set requirements and guide the product direction. 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. To support this, we run in parallel a supporting course on Software Engineering *Practice*, covering development methods, tools, quality assurance, project and product management techniques, all aimed at enhancing the team-working experience and maximising each team's productivity and chances of success.

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 an agile method - either Extreme Programming, Scrum, or Kanban - and back this up with engineering practices such as continuous integration, automated testing and staged deployments. While these 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.
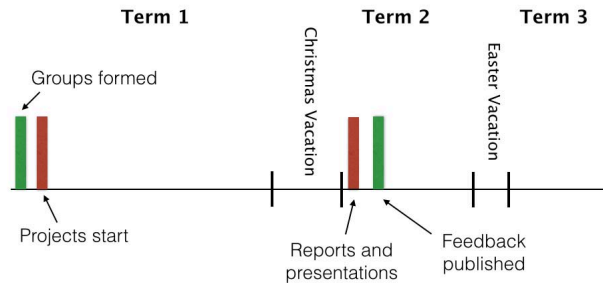
Fig. 5. Schedule for third year Group Projects - project duration is 3 months

Fig. 6. Perceived effort curve for students during Group Projects (in blue)

We have found the Software Engineering Practice course to be one of the most difficult to get right. The main problem is one of relevance and application. We want the material to support the projects and enable students to deliver better products as a result. However, there is a danger that teaching and assessing software engineering practice takes time away from the group work itself. Furthermore, although group projects are similar in format they differ widely in the technical challenges that need to be overcome. For example, some may be developing mobile apps, while others create web applications, desktop software or even command line tools.

In an early version of this course, we delivered a series of lectures and required students to write up reports on how techniques for project management, quality assurance and particular technologies could be applied to their projects. These reports were assessed separately from the group project itself. The problem we found was that whilst a particular tool or technique – for example cloud deployment – may be perfectly suited to some projects, it may be irrelevant to others. As a result many students were either not motivated to spend time on topics that did not directly apply to them, or felt aggrieved that learning and being assessed on such topics took away precious time from their development work, as exemplified by the following survey comments:

> "[writing reports about tools and processes] (at times) felt like it took away time I'd have liked to spend on actually working on the project."

> "I think you need to change the [course] structure somehow, it took too much focus away from the actual projects."

Another problem that we identified is that although we are encouraging the students to adopt agile methods, and to work in an iterative way, the project assessment, which was separate from the assessment of software engineering practice, was more in line with a waterfall model - a one-shot delivery in January, comprising the developed software, an associated project report, and a presentation. There was no particular incentive for students to work at an even pace throughout the term, and project work was often put aside in favour of coursework for other courses. As the deadline was not until after Christmas, there was no great urgency to make
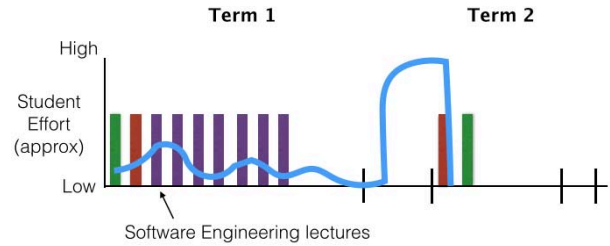
progress early in the term. Based on talking to groups and supervisors anecdotally, our impression was that the students' effort on project work during the term roughly followed the curve shown in Fig. 6.

In response to feedback from students and supervisors, and aiming to provide a tailored learning experience relevant to each project team, the following year we cut back the assessment for software engineering practice radically. We also reduced the number of lectures and offered supporting material online for students to refer to on demand to address issues in their projects. To support this we introduced consultation time for each group through "clinic hours" that they could use at will. This seemed an attractive structure, as by allowing groups to pull help when they needed it, we ought to be optimising for relevance, and eliminating waste from the system.

Unfortunately this did not work very well. Many students chose not to pull help or to dig in to the online material. They perceived that they were better off spending their time "getting on with the project". To an extent they did not know what they did not know. To add to this, the size and length of the project was probably not great enough for them to feel pain caused by lack of software engineering practice until late on - perhaps during the effort spike just before the deadline (Fig. 6). Anecdotally, we found that those groups that did use the consultation sessions generally produced better projects, but may be down to self-selection: better organised groups, or groups containing a higher proportion of high-achievers, tended to be the ones that made use of the clinics.
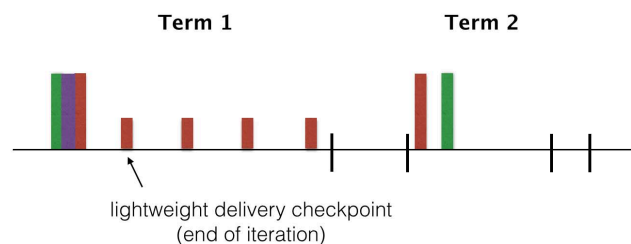

Fig. 7. Lightweight end-of-iteration checkpoints every two weeks

In order to encourage development at a consistent and sustainable pace throughout the project without burdening

students and supervisors with assessment we have recently (from 2016) introduced bi-weekly checkpoints, effectively enforcing a two-week iteration cadence in a similar spirit to the weekly assessment cycle in Software Engineering Design (Section IV). We have given marks – the only real currency that we have – for demonstrating enhanced functionality every two weeks. To acquire the marks, each group is required to demonstrate progress, and useful, working software, to their supervisor, who will give their sign-off if they are satisfied. It is important that the teams deliver something simple but end-to-end in early iterations. For example, there are no marks for a beautiful database schema with no user interface. Through this mechanism we encourage a pattern of development that maps closely to the relationship between the customer and the development team expected in a mature agile team.

As before, following this pattern gives us a mechanism to give feedback more often and in smaller batches. In order for this mechanism to work, we have made the deliverables lightweight – not a project progress report, just a demonstration of the software produced, a customer conversation, and a signature on a piece of paper.

The software engineering practice material now comprises a short period of intensive tuition at the beginning of the project covering methods and tools that all teams could use to manage their work and ensure the quality of the software they build. Once the two-week iterations are under way, we again offer clinic hours for individual consultation to help with issues arising, where we can address the problems of individual teams in depth. The later iterations of these projects revolve around adding functionality to each component of the initial "walking skeleton" to grow the system towards maturity.

As the group projects have evolved we have moved away from teaching and assessing theoretical knowledge and understanding of agile methods per se, preferring instead for students to learn by experience. Where some other universities have simplified the technical deliverables of their projects to allow students to focus on learning the processes [20], we have instead tried to simulate industrial conditions as closely as possible, focussing on regular delivery of working software.

VI. Software Engineering for Industry

The last aspect of software engineering that we address in our programme, with the aim of producing industry-ready engineers, is an exploration of industrial practice and experience. The aim is to discuss issues that are current hot topics in the industry, and to look at wider forces that affect software engineering decisions, particularly those that are economic and social as well as those that are technical. We strive to keep this course up to date with current industrial trends, drawing on practical experience, whilst keeping a grounding in fundamental engineering principles.

The syllabus is fluid, but an important recurring theme is working effectively with legacy code [21]. A large proportion of practising software engineers spend their working lives making changes to existing codebases, rather than starting from scratch. Successful systems evolve [22] and need to

be updated as new requirements come in, market conditions change, or other new systems need to be integrated. This is not second class work, but engineers need techniques to work in this way which differ from what they might do if they had free reign to start from a blank slate. How to comprehend an unfamiliar codebase? How to manage risk when making changes? When is it better to refactor, and when to rewrite?

Other topics the course has covered recently include software evolution, microservices, continuous delivery, and resilience at scale. Such topics are the realm of opinion rather than hard fact. Our aim is for students to develop their critical thinking, and to voice their own opinions and arguments based on reading around each topic presented. There is thus a research element to the course and this exercises the students' ability to use reasoned engineering judgement. As the course has evolved, we have delivered less content by transmission, and have instead designed an experience where students can participate and learn for themselves.

It is vital that this course is facilitated by people with industrial backgrounds; previous experience has taught us that students are very quick to detect a lack of authenticity among the contributors. To strengthen the industrial perspective, each week we invite a "panel" of industrial practitioners as guests – normally two per week. We elicit the panel's views on the topic under discussion, and they share their own stories and examples during the classroom discussions. Note that this is not the same as simply including a couple of guest lectures within a lecture course. We carefully plot out the narrative arc of the course, and then invite guests who can illustrate each topic based on their experience, drawing on contacts in local companies. In particular, we make sure that every guest knows what topics have been covered in earlier weeks of the course, what topics are coming up, and the message that we are aiming to deliver. We have found that alumni can make excellent contributors, provided they are well versed in the format and objectives of the discussion sessions. The guest speaker model in this course has worked exceptionally well, as highlighted in many survey comments, for example:

> "I thoroughly enjoyed the guest speakers, they offered real world problems that they had solved utilising the course material. It was also great to hear a lot of discussion with the guests in [class]."

A. Format and Assessment

The current course format combines reading, coding, writing and discussion. We examine a new topic each week for seven weeks, and the main part of the week's work is undertaken outside of the classroom. Students research the topic through blogs, articles, papers, videos of conference talks etc. and are required to write a short position statement based on this, answering one of a given set of discussion questions. The classes involve a series of discussions which are initiated by selected students briefly presenting their findings from their week's work. It is important that the deadline for the written work is *before* the discussion class. This means that the facilitator can assume that every student present has

done some reading and thinking about the topic. They can then direct the discussion based on the content of the written work and are relatively free to call on any student to contribute.

As with our experience in the second and third year courses, we have found this course to be most successful when the students work in weekly submission cycles. There is no final exam, so all the marks are accrued from the weekly courseworks together with a small number of more open-ended assignments. The solution to the weekly marking load for this course was not to add automation, but to change the constraints on the learning experience to cause the students to learn from their discussions with one another, and to reduce the volume of work submitted for assessment. To do this, we changed from individual work, to paired work, to groups of three – in each case reducing the number of submissions and increasing the number of ideas and viewpoints to be incorporated. We also imposed a relatively short word limit of 300 words per submission. This meant that submissions cannot be long and meandering; they must be short and well argued, making them much easier to assess. At the same time it increases the amount of discussion the students must have refining their ideas and constructing their arguments, as exemplified by the following survey comment:

> "The quick feedback on the [assignments] was extremely appreciated and allowed us to implement the suggestions for the next exercises. The questions were challenging and forced us to spend hours thinking about pros and con and to do extensive research on the topics."

B. Evolving Practice

As we have iterated on the second and third year courses, we have tried to include more and more industry-relevant content, and this has often meant moving material down from the fourth year course. For example some material on test-driven development that we used to cover in the fourth year is now a core part of the second year, and an introduction to agile methods is now a key feature of third year software engineering practice course that supports the group projects. While we do not want to be jumping on all the passing trends, this advanced course gives us a vehicle to discuss and distil the current state of practice, and to filter ideas down into lower years whenever they become core. We therefore believe that it provides an excellent driver for ensuring that our software engineering material is relevant and up to date at all levels. It also instils an important element of critical thinking and analysis that employers often look for at interview. The students' own experiences in this respect have proven to be very positive, for example:

> "The course is very well structured and you learn things that are really useful in the industry! I can say that this course helped me to get a job offer; every single thing that we have learnt during this course was asked during the interview process."

> "Overall I'd say the Software Engineering for Industry course was one of the most enjoyable this year...

Ultimately, software engineering courses and frequent group projects are what helps make Imperial students so very employable – this course embodies that success."

VII. DISCUSSION AND LESSONS LEARNT

While we could try to present improvements in the exam grades achieved by classes over the years as we worked on different aspects of these courses, we do not feel that this makes for a meaningful comparison, as there are too many variables affecting students' results. We have a different cohort of students each year and we cannot put the same class of students through two different variants of the same course in a properly controlled experiment. Although the classes are large, with approximately 150 students in each year group, it is hard to contend that quantitative results would be statistically significant. Instead we will discuss the lessons that we have taken from the iterative improvements to the format and content of the programme and constituent modules, and the learning outcomes achieved.

Decreasing batch size for feedback increases engagement. By setting assessed work each week, we encouraged students to participate actively in their learning in a sustained way throughout modules. Even a small amount of credit available motivates them to work on weekly tasks, where previously many students would not have attempted unassessed exercises, or left them until much later as revision aids. Students greatly appreciate getting weekly feedback on their work so that they can check their progress, and determine where they need to focus their efforts to meet the expected learning outcomes for the course. We saw this in both the second year and fourth year courses, and are currently implementing the same mechanism to try to flatten the effort curve in third year software engineering group projects.

Setting many small targeted assignments, rather than larger projects, helps increase consistency in learning outcomes across a class. When we gave larger, more synoptic coursework projects, we found that the results showed a lot of variation in the skills and knowledge that students could demonstrate. By leading them through a more structured programme of smaller, more focussed exercises, incentivised by small amounts of credit, we made sure that every student had done in depth study on each of the core topics throughout the course. In the second year, this manifested itself in every student having implemented a core set of design patterns, and having developed their automated testing and refactoring skills by applying these to every exercise. In the fourth year, the weekly assessments led to all students investing time researching and discussing every topic in our syllabus.

Investing time in tooling and automation supports tighter feedback loops. Decreasing batch sizes and setting more, smaller, assignments increases the burden substantially in terms of giving feedback. Although ideas of automating this entirely are probably wishful thinking – at least if insightful and nuanced comments are to be made – we have found that by developing tooling, and integrating it into the students'

workflow, we can provide basic feedback automatically, and make better use of the markers' time. We have found that tools built for one class can be reused in other classes, and in subsequent years, with only minor modifications – amortising the development effort required to build them. For programming-based courses, we also found that managing all of the students' work using a version control system gave them familiarity with tools that they would use daily in industry.

Aiming for sustained opportunities for students to practise skills, and short feedback cycles, caused us to change class formats in a way that also supported more developmental learning. In our attempts to make short feedback cycles more tractable, we introduced pair-programming and group work. One of our primary drivers in doing this was to reduce the number of submissions that had to be processed each week, but a very positive side-effect was that it made the learning experience more developmental, increasing discussion and debate between fellow students, so that they ended up coaching one another. A similar effect was produced when we set a relatively small word limit (just 300 words) for written work by our fourth year groups. Shorter submissions were quicker to read and to mark, and also caused the groups to spend longer discussing and distilling their arguments and setting out their ideas clearly and succinctly - again, learning from their interactions with one another.

Engaging practitioners with industrial backgrounds improves the relevance (and perceived relevance) of course material. By nurturing links with practitioners we have been able to adapt our curriculum to align with the needs of the industry. We have adopted industrial tools in student projects and assignments, and constructed practical courses teaching techniques like test-driven development and working with legacy code that reflect industrial practice, while maintaining our focus on the principles of computer science. Bringing practitioners into the classroom, either as guest lecturers, panellists, or by creating roles where they can join the staff, has given a credibility to the material being taught, and an extra level of insight drawn from experience which resonates deeply with the students looking forward to their future careers.

Evidence suggests that we are producing graduates who are well versed in modern software engineering principles and practices, and whose skills align well with the needs of industry. Our undergraduate degrees currently top LinkedIn's university rankings [23] for career outcomes of graduate software engineers in the UK and our graduates' starting salaries in these roles are impressive [24]. Furthermore, feedback from students, which has in part driven the evolution of the programme, also suggests that the skills that they learn are of great benefit to them, both in obtaining sought after jobs and in becoming better and more productive engineers.

REFERENCES

[1] M. Kropp and A. Meier, "New sustainable teaching approaches in software engineering education," in *2014 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2014, pp. 1019–1022.
[2] C. Anslow and F. Maurer, "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*. ACM, 2015, pp. 500–505.
[3] E. Papatheocharous and A. S. Andreou, "Empirical evidence and state of practice of software agile teams," *Journal of Software: Evolution and Process*, vol. 26, no. 9.
[4] Association of Computing Machinery, "Computer science curricula 2013," online, 2013. [Online]. Available: https://www.acm.org/education/CS2013-final-report.pdf
[5] K. Beck, *Extreme Programming Explained: Embrace Change*, ser. An Alan R. Apt Book Series. Addison-Wesley, 2000. [Online]. Available: https://books.google.co.uk/books?id=G8EL4H4vf7UC
[6] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.
[7] D. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010. [Online]. Available: https://books.google.es/books?id=RJ0VUkfUWZkC
[8] J. Womack and D. Jones, *Lean thinking: banish waste and create wealth in your corporation*, ser. Lean Enterprise Institute. Simon & Schuster, 1996. [Online]. Available: https://books.google.co.uk/books?id=DJwoAQAAMAAJ
[9] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.
[10] M. Guzdial, *Learner-Centered Design of Computing Education: Research on Computing for Everyone*, ser. Synthesis Lectures on Human-Centered Informatics. Morgan & Claypool Publishers, 2015. [Online]. Available: https://books.google.co.uk/books?id=BAlVCwAAQBAJ
[11] E. Alpay, P. S. Cutler, S. Eisenbach, and A. J. Field, "Changing the marks-based culture of learning through peer-assisted tutorials," *European Journal of Engineering Education*, vol. 35, no. 1, pp. 17–32, 2010.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
[13] S. Metz, *Practical Object-Oriented Design in Ruby: An Agile Primer*. Pearson Education, 2012. [Online]. Available: https://books.google.co.uk/books?id=VRCv_bATuSIC
[14] M. Rother, J. Shook, and L. E. Institute, *Learning to See: Value Stream Mapping to Add Value and Eliminate Muda*, ser. A lean tool kit method and workbook. Taylor & Francis, 2003. [Online]. Available: https://books.google.co.uk/books?id=mrNIH6Oo87wC
[15] T. Ottinger, "Over-starting and under-finishing," online, 2015. [Online]. Available: https://www.industriallogic.com/blog/over-starting-and-under-finishing/
[16] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.
[17] M. Fowler, "Frequency reduces difficulty [online]," July 2011. [Online]. Available: http://martinfowler.com/bliki/FrequencyReducesDifficulty.html
[18] L. Williams, E. Wiebe, K. Yang, M. Ferzli, and C. Miller, "In support of pair programming in the introductory computer science course," pp. 197–212, 2002.
[19] B. Jones, R. Sodhi, M. Murdock, R. Mehra, H. Benko, A. Wilson, E. Ofek, B. MacIntyre, N. Raghuvanshi, and L. Shapira, "Roomalive: Magical experiences enabled by scalable, adaptive projector-camera units," in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '14, 2014, pp. 637–644.
[20] J.-P. Steghöfer, E. Knauss, E. Alégroth, I. Hammouda, H. Burden, and M. Ericsson, "Teaching agile: Addressing the conflict between project delivery and application of agile methods," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, 2016, pp. 303–312.
[21] M. Feathers, *Working Effectively with Legacy Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.
[22] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94, 1994, pp. 279–287.
[23] N. Kapur, "Ranking universities based on career outcomes [online]," Nov 2014. [Online]. Available: https://blog.linkedin.com/2014/10/01/ranking-universities-based-on-career-outcomes
[24] Unistats, online. [Online]. Available: http://unistats.direct.gov.uk/Subjects/Overview/10003270FT-G401/ReturnTo/Search