Teaching Modern Software Development Techniques at University

Key takeaways

- People often perceive a large gap between what is taught in university and the “real world”.
- Practitioner input into university degrees can help keep them contemporary.
- Use of modern tools and techniques in courses helps with relevance and engagement.
- We teach software design through TDD and refactoring rather than up front specification.
- Comparing research papers with industrial blog posts and articles yields interesting discussion of the state of the art.

On completing a computer science degree, a large proportion of graduates proceed into industrial jobs in software development. They work in teams inside organisations large and small to produce software that is central to business and in many ways underpins the daily lives of everyone in the developed world.

development projects. We often hear how there is a skills shortage in the software industry, and about the apparent gap between what people are taught in university and the "real world". In this article we will explain how we at Imperial College London have developed a programme that aims to bridge this gap, providing students with relevant skills for industrial software engineering careers. We will also describe how we have tried to focus the course around the tools, techniques and concerns that feature in the everyday life of a professional developer working in a modern team.

Classes in universities are almost always taught by academic researchers, 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. Similarly 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.

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 which 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.

constituent courses. This has ranged from getting individual pieces of advice on current issues, helping to outline course content, having practitioners come in to give guest lectures or coaching, to - in my own case - joining the staff. We have found that practitioners are generally very happy to help us shape the curriculum for the next generation of software engineers, to give something back to the community, and of course helping with teaching can also be an opportunity to promote their companies if they are recruiting.

Course Content

At Imperial we have a three or four year programme in Computing. Students can study three years for a BEng degree, or study an extra fourth year and receive an MEng degree. The first three years are fundamentally the same for both programmes, but those going on to take the fourth year also do a six month work placement with a company between their third and fourth years. Here we will describe the core modules that we feel constitute the “software engineering” element of the course - although alongside these, students study modules in mathematics, logic, compilers, operating systems and many other aspects of what might be thought of as “computer science”.

First Year

In the first year of the degree programme, we concentrate on basic programming skills. We believe that these are fundamental for all of our students, and these are taught through lecture courses in functional, object-oriented and systems programming, supported by integrated computer-based laboratory exercises. The lab exercises are very important, as it is through these that students get to practise programming, get personalised feedback, and improve.

need to support both of these groups in our introductory course - not making the inexperienced coders feel like they are disadvantaged, whilst not boring the more experienced students with material they already know. The main thing we have done to try to level the playing field is to start by teaching Haskell as the first language. This is usually equally unfamiliar to almost all of the new students - even those who have programmed a lot typically have not used this type of language before.

One innovation that has proved very successful is introducing the use of version control right from the very first week. Rather than being “a tool for collaborative projects” used later on, we have made it so that every lab exercise involves cloning a starting point from a git repository, making incremental commits, and then what the students submit for assessment is a git commit hash pointing to the version that they want marked. This makes use of version control something that is completely natural and an everyday activity.

Second Year

In their second year, we aim to teach students how to design and develop larger systems. We want to move on from teaching programming in a particular language, and to look at larger design concerns. In an earlier iteration of this course, the material concentrated on notation, specification languages and catalogues of design patterns. This meant that students would know a range of ways to document and communicate software designs, which were not tied to a particular implementation language. However, when comparing this course content with the design practices predominantly used in industrial projects, we found some mismatches.

support business in different types of enterprise, consumer web services, apps and games etc. The use of formal specification techniques amongst these sorts of teams is relatively rare. Also, as agile development methods are now common, 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.

We still give students the vocabulary of design patterns and architectural styles, but with each we look at the problem it is aiming to solve (for example the removal of duplication) and any trade-offs that may apply (for example introduction of coupling caused by the use of inheritance, and how this might affect future changes). We have moved towards grounding the examples in code, accompanied by tests, and cast design changes as evolutions and refactorings affecting various qualities of the codebase that we are working on. By working concretely with code, we have found that students engage more directly with different design concerns, and the effects of the forces as play in the system, than they did when thinking about designs more abstractly. We can use modern IDEs to manipulate code into different structures, use metrics to talk explicitly about code quality, complexity, coupling etc, and the students can learn kinaesthetically by working through problems and producing practical solutions.

Third Year

particular problem or provides a certain service for their users. Each group has a customer - either a member of the faculty, or an industrial partner, to guide the product direction. The main aims of this project 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 course on Software Engineering Practice, covering development methods, tools, quality assurance, project and product management techniques, etc.

This has been one of the most difficult courses for us to get right. The main problem is one of relevance. We want the software engineering course to support the group projects, and for the two to be integrated. But, feedback from the students has often been that they felt that the software engineering course was a distraction, and that they would prefer to spend time "just getting on with the project". This shows that students were not feeling the benefits of the taught software engineering practices in their own projects. We considered two possible reasons for this.

Firstly, the range of projects being carried out by different groups is wide. Some may be developing mobile apps, while others create web applications, desktop software or even command line tools. If we include material in the curriculum about a particular topic that is relevant to some project groups - for example cloud deployment - it may be irrelevant to others. The more content we try to include in the course, the greater the chance that we are asking students to spend time learning a topic that they feel does not affect their project.

of time, integrating many different aspects. We encourage them to set up collaboration tools and procedures to help them work together both in terms of technical code and software management, and also more general project management. At the beginning of the project, these can seem like overhead - especially the time spent setting up tools, which again can seem like time lost from "getting on with the project". It is only towards the end of the projects, when pressure is on to deliver, that these tools and techniques return rewards on that investment.

Fourth Year

The final part of our four year programme is a course entitled Software Engineering for Industry. The main philosophy behind it has been to give students a view of some of the issues facing industrial software engineers, essentially preparing them for the world of work. 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 done in the third year to support 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 distill the current state of practice, and to filter things down into lower years once they become core practices.

One of the main topics that we aim to cover in this course is working effectively with legacy code. A large proportion of practising software engineers spend their working lives making changes to existing codebases, rather than starting from scratch. This is not a bad thing, it is normal. Successful systems evolve 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. When might it be more appropriate to refactor, and when to rewrite?

week's work is to research the topic through blogs, articles, papers, videos of conference talks etc and to write a short position statement based on this answering one of the discussion questions. Then we have a discussion class where students briefly present and discuss their findings from their week's work. To add to the industrial viewpoint, each week we invite a “panel” of industrial experts as guests. We elicit the panel's views on the topic under discussion, and they bring their own stories, examples and case studies to share. As we develop the course, it feels less like we are delivering content, and more designing an experience through which the students can participate and learn for themselves.

Perspectives on Teaching

Not all material is taught in the same way, and we are continually trying to improve the learning experience. One way that helped to think and talk about this was to consider the different ways that students learn in terms of three perspectives described by Mark Guzdial in his recent book Learner-Centred Design on Computing Education. Guzdial characterises different learning experiences as Transmission - the transferral of knowledge through a one-way medium like a lecture, Apprenticeship - where students focus on developing skills by practising them in exercises, and Developmental - where each student gets individual help with the things that will help them personally to advance, not necessarily aligned with the rest of the class.

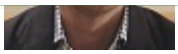
In teaching software engineering, we still have quite a lot of transmission (even though there is evidence [http://www.pnas.org/content/111/23/8410] that it is not so effective, tradition is hard to overcome), but we are starting to focus more on apprenticeship models and the deliberate practice of skills, particularly in terms of software development. It is hard to give students frequent one-to-one attention with class sizes of 150 students, and relatively few tutors, but as we encourage more group work and particularly pair programming in student assignments, we find that students are able to coach and learn from each other, getting individual developmental help from their peers. Prof Laurie Williams at NCSU has done a lot of work showing the effectiveness of pairing programming in teaching. [http://collaboration.csc.ncsu.edu/laurie/pair.html]

approach has been to think about the delivery of ideas as a value stream. If we start with a big list of requirements for what students should learn (a syllabus) and then over the course of a few months, transmit them via lectures, and at the end perform some quality assurance on this learning by giving the students an exam, then we have something that feels very much like a waterfall development process. In software development, the industry has evolved to value fast feedback and frequent delivery of value in small batches. Can we work towards the same goals in iterating on our learning experiences?

One thing we have done along this path is introducing weekly, small assignments, rather than big end-of-term assessments. This encourages students to work at a more sustainable pace across the term, and gives them and their tutors feedback on how well they have understood each concept. For example, in our software design module, we aim to have a targeted practical exercise each week, so that students can practise a particular aspect of design by writing code and tests, and get feedback on their work within a few days. Of course this generates a large load on the tutors to mark and return a large number of assignments in a short cycle. It is tempting to relent, and reduce it to fortnightly, or monthly assignments, but again following the principles we would apply in an agile project, we have tried to use automation to give initial feedback early, and make the work of the human marker easier, so that it can be done more often. We are not there yet, and there is still a lot of work for the tutors to do each week to give good quality feedback, but it feels like we are heading in the right direction.

We still have lots of problems to solve, and the constantly changing state of the software industry means that we will have to constantly update our curriculum to stay relevant, balancing computer science fundamentals (which hopefully do not change that often) with industrial trends and the application of modern tools and techniques. But, as we would if we were running a software project, we hope to continue to inspect and adapt and continuously improve.

About the Author



companies from startups to multinationals, including working as an engineer at Google, and also as a technical lead at Kizoom, one of the earliest companies in the UK employing XP at scale. He has chaired the XPDay conference, and acted as programme chair for the SPA conference. Robert holds an MEng degree in Information Systems Engineering and PhD in Software Engineering from Imperial College London.

4