Software Engineering Curriculum Technology Transfer:

Lessons learned from Ebooks, MOOCs, and SPOCs

Armando Fox
UC Berkeley
fox@cs.berkeley.edu

David Patterson
UC Berkeley
pattrsn@cs.berkeley.edu

Abstract

This paper describes our experience in trying to transfer our revised software engineering curriculum from UC Berkeley to other universities. Our original plan was just to develop an inexpensive electronic textbook, but we were swept up in the first wave of Massive Open Online Course (MOOCs) while we were writing it. Thus, the paper lists the lessons learned about educational technology transfer from writing Ebooks and developing MOOCs. To make it easier for instructors to use MOOC material, EdX offers Small Private Online Course (SPOCs) We argue that SPOCs and Ebooks may become the textbook of the 21st Century.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.9: Management; D.2.10: Design; K.3.2 [**Computers And Education**]: Computer and Information Science Education

General Terms Management, Documentation, Design.

Keywords Agile development; cloud computing; education; electronic book; massive open online course; software as a service.

1. Teaching Software Engineering: Six Challenges

As being a software engineer is one of the most attractive jobs in the country [1], undergraduate students are understandably eager to learn software engineering. Within a computer science or computer engineering department, that material is typically taught in a one semester course or in one or two quarter courses.[1] As students take typically four courses at time, if we assume a 50 hour week, that leaves between 1*15*50/4 to 2*10*50/4 or 190 to 250 hours per course. The first challenge is that students have just five to six full-time weeks to be introduced to a topic as vast as software engineering!

The second challenge is that it is unlikely that the faculty teaching the course are practicing software developers, nor in most cases do they do research in software engineering. Thus, they are usually not experts in what they must teach.

A third challenge is that there are many software development methodologies from which to choose. While it makes sense to survey many of them to familiarize students with the options, there are obvious advantages to picking a single one for students to use on projects, for example, so the staff can answer questions and find documentation to help students practice the chosen methodology. It is not easy to know which one to pick.

A fourth challenge is that software engineering textbooks are primarily surveys of software problems and descriptions of the many development methodologies for many platforms. Such surveys are unsatisfying in part because they usually don't go into enough detail in any methodology to be able to follow it, and in part because it is hard to decide which one to use. Reviews of the most popular textbook in software engineering, first published in 1982 and now in its seventh edition, illustrate this dissatisfaction [2]: its average quantitative reviews at Amazon.com are 1.7 on a scale of 5, and few authors would enjoy the comments highlighted for this book. The lack of good textbooks to help instructors prepare lectures and help students learn on their own adds to the teaching challenge.

A fifth challenge is that the tools to support many methodologies are either lacking or too expensive to be deployed in a college course. The lack of tools makes it hard both for students to follow the advice in lecture *and* for instructors to check to see if the advice is being followed.

A final challenge, in part resulting from the first five, is that industry commonly complains about the quality of software engineering education. We can't think of another CS course that is routinely lambasted by employers of our graduates, which is ironic since it is arguably one of our most important courses.

The result is that instructors try to lecture about software engineering topics, but students continue to build software more or less the way they always have; thus, the software engineering course in practice is often nothing more than a project course. The faculty reward for agreeing to teach this important topic is often poor teaching evaluations from their students. This sad but stable state of affairs is frustrating to instructors, boring to students, and disappointing to industry.

Fortunately, there is a path that addresses all six challenges.

2. Teaching Software Engineering Agilely[2]

While one of us developed software part time for a local theater as a volunteer, and thus was familiar with recent trends in software development, neither of us were researchers in software engineering. Hence, we considered ourselves novices when preparing to teach a software engineering class.

Thus, our first step was to speak to representatives from a half-dozen leading software companies to understand their complaints

SPLASH Education Symposium, October 28, 2013, Indianapolis, IN, USA

[1] While a few programs offer a Software Engineering degree, where students take a half-dozen such courses, the vast majority of students are getting CS or CE degrees.

[2] This section is derived from Fox and Patterson [3] and Fox and Patterson [4].

about how software engineering is taught. We were struck by the unanimity of the number one request from each company: that students learn how to enhance sparsely-documented legacy code. In priority order, other requests were making testing a first-class citizen, working with non-technical customers, performing design reviews, and working in teams.

We were already planning for students to do projects in teams, which addressed one of industry's requests. To gain experience in working with non-technical customers, we recruited proposals from nearby non-profit organizations. They proved to be an excellent resource, as non-profits had modest budgets for information technology and thus welcomed the help provided by teams of computer science students.

The ACM-IEEE Joint Task Force on Computing Curricula 2013 [5] later confirmed the wisdom including team projects:

> *In general, students learn best at the application level much of the material defined in the [software engineering knowledge area] by participating in a project. Such projects should require students to work on a team to develop a software system through as much of its lifecycle as is possible. Much of software engineering is devoted to effective communication among team members and stakeholders. Utilizing project teams, projects can be sufficiently challenging to require the use of effective software engineering techniques and that students develop and practice their communication skills. While organizing and running effective projects within the academic framework can be challenging, the best way to learn to apply software engineering theory and knowledge is in the practical environment of a project.*

2.1 Picking a Platform and Methodology

A software project must target some platform and make use of some development methodology. We decided to pick the platform and methodology that had the best set of programming tools, for three reasons:

1. Students were much more likely to follow a methodology if there was a tool that made it easy for the students to do so.
2. If the staff is able to grade the output of the tool, we could evaluate the *intermediate* stages of the development process, not just the final project. Such in-process grading inspires students to follow the advice in the lecture.
3. Given that there are only five to six fulltime weeks to learn this important field, we hoped that the productivity gains from the tools would allow students to spend their effort on higher-level issues of the project.

To motivate students to work on their projects, it's helpful to use a platform that lets them create compelling apps. In this Post-PC Era, mobile applications for smart phones/tablets and Software as a Service (SaaS) for cloud computing are both compelling.

Software development methodologies can be divided into two camps:

1. *Plan-and-Document*. These methodologies try to make software development more predictable via careful planning and extensive documentation. Examples are waterfall, spiral, and the rational unified process.
2. *Agile*. Rather than rely on plans and documentation, this approach embraces change as a fact of life; small teams of developers continuously refine a working but incomplete prototype until the customer is happy with result, with the customer offering feedback each iteration, which are frequent. Examples include extreme programming and scrum.

Although the Agile Manifesto was considered controversial when released in 2001, Agile is an accepted practice today. A recent survey of 66 large software projects in industry found that the majority used Agile[6], and the latest editions of the most popular software engineering textbooks now introduce Agile early [2][7].

2.2 SaaS and Rails

We found that the tools for Agile development of Software as a Service for cloud computing had by far the best tools, in particular the Ruby on Rails ("Rails") programming framework.

Agile emphasizes *Test-Driven Development*[3] (TDD) to reduce mistakes, which addresses industry's request to make testing a first-class citizen; *user stories*[4] to elicit and validate customer requirements, which aids in working with non-technical customers; and *velocity*[5] to measure progress. The Agile software philosophy is to make new versions available every one or two weeks. Clearly, small teams and multiple iterations of incomplete prototypes sound like a good match to the classroom.

The Agile assumption is basically continuous code refactoring over its lifetime, which develops skills that can also work with legacy code. Finally, to address our industrial colleagues number one request, we have a programming assignment where students use their Agile skills to enhance legacy code.

Once again, the ACM-IEEE Joint Task force later affirmed our choice [5]:

> *... there is increasing evidence that students better learn to apply software engineering approaches through an iterative approach, where students have the opportunity to work through a development cycle, assess their work, then apply the knowledge gained through their assessment to another development cycle. Agile and iterative lifecycle models inherently afford such opportunities.*

To do multiple iterations in a single course—we do four iterations at UC Berkeley—they must be just one or two weeks in length, which suggests Agile development. Indeed, with Agile students have the "space" to make mistakes, analyze them, and make improvements for the next iteration throughout the entire course.

SaaS and cloud computing also simplifies the management of the course. Students can deploy their projects using the same horizontally-scalable environment used by professional developers, which is instant, free for small projects, and requires neither software installation nor joining a developer program. In particular, it separates the course from instructional computers, which are often antiquated, overloaded, or both.

2.3 Cucumber Tool: From User Stories to Acceptance Tests

The Rails ecosystem has by far the best tools to support test-driven development, behavior-driven design, and Agile processes, many of which are made possible by intellectually deep Ruby language features such as closures, higher-order functions, functional idioms, and metaprogramming. Because these tools are lightweight, seamlessly integrated with Rails, and require virtually no installation or configuration—some are delivered as SaaS—students quickly learn important techniques by doing them.

Our experience has been that the extra time in the class to teach Ruby and Rails—as opposed to trying to teach the class using languages and tools they already use—is more than paid back in the productivity gains from the Rails tools that they sub-

---

[3] In TDD you first write a failing test case that defines a new feature, and then write code to pass that test

[4] A user story is a few nontechnical sentences that capture a feature that the customer wants to include in the app.

[5] Velocity is calculated by estimated units of work per user story and then counting how many units are completed.

sequently use. Compared to Java and its frameworks, Rails programmers have found factors of 3 to 5 reductions in number of lines of code, which is one indication of productivity.[8,9] Picking up a new language, framework, and tools has the added benefit of more realistically demonstrating the lifelong learning expected from software engineers.

For example, the Cucumber tool turns the user stories from the non-technical customer into acceptance tests for the app. As a result, it *rewards* students who follow the user story methodology rather than having requirements elicitation feel like just another bothersome burden that faculty foist on their students in software engineering courses.

Below is an example feature for a cash register application and one "happy path" user story (called a *scenario* in Cucumber) for that feature [10]:

```
Feature: Division
  In order to avoid silly mistakes
  Cashiers must be able to calculate a fraction

  Scenario: Regular numbers
    Given I have entered 3 into the calculator
    And I have entered 2 into the calculator
    When I press divide
    Then the result should be 1.5 on the screen
```

Note that this format is easy for the non-technical customer to understand and help develop, which is a founding principle of Agile. It also addresses a criticism from industry. Cucumber uses regular expressions to match user stories to the testing harness. Below is the key section of the Cucumber and Ruby code that automates the acceptance test by matching regular expressions:

```
Given /I have entered (\d+) into the calculator/ do |n|
  @calc.push n.to_i
end

When /I press (\w+)/ do |op|
  @result = @calc.send op
end

Then /the result should be (.*) on the screen/ do |result|
  @result.should == result.to_f
end
```

Such tools not only make it easy for students to do what they hear in lecture, but also simplify grading of student effort from a time-intensive subjective evaluation by reading code to a low-effort objective evaluation by measuring it. Cucumber shows the number of user stories completed, and Pivotal Tracker records weekly progress and can point out problems in balance of effort by members of teams. Indeed, these tools make it plausible for the online course (see Section 4) to have automatically gradable assignments with some teeth in them. Other ready-to-run open-source tools measure test coverage, cyclomatic complexity [11], assignment-branch-condition complexity [12], and code smells. We provide a Virtual Machine image preloaded with all these tools and deployable on the free VirtualBox hypervisor or on Amazon's Elastic Compute Cloud.

The net effect of this course is to move students out of their "comfort zone." Throughout their undergraduate education they are assigned tasks and projects for which they are given complete specifications, for which there are complete and known solutions, and for which they program on familiar platforms in the same small set of familiar languages. For the most part, this is exactly what they *won't* find after graduation. This is the rare course where students must derive their own analysis and specification of a project requested by a customer, and where they may be required to develop software on an unfamiliar platform using an unfamiliar language and tools for which there is no pre-derived solution.

2.4 Addressing Criticisms of Agile and Rails

Rails also helps with a criticism of Agile in that TDD and rapid iteration can lead to poor software architecture. Indeed, the Rails framework follows the Model View Controller (MVC) design pattern to simplify development of the classic three-tiered applications of cloud computing.

One criticism of the choice of Ruby is its inefficiency compared to languages like Java or C++. Since hardware has improved roughly 1000X in cost-performance since Java was announced in 1995 and 1,000,000X since C++ was unveiled in 1979 [13], the efficiency of low-level code matters in fewer places today than it used to. We think using the improved cost-performance to increase programmer productivity makes sense in general, but especially so in the classroom.

Note that for cloud computing, horizontal scalability can trump single-node performance; deploying SaaS on the cloud in this course lets us teach (and test) what makes an app scalable across many servers, which is not covered elsewhere in our curriculum. By using the cloud to teach the class, we can offer students the chance to experiment with scalability.

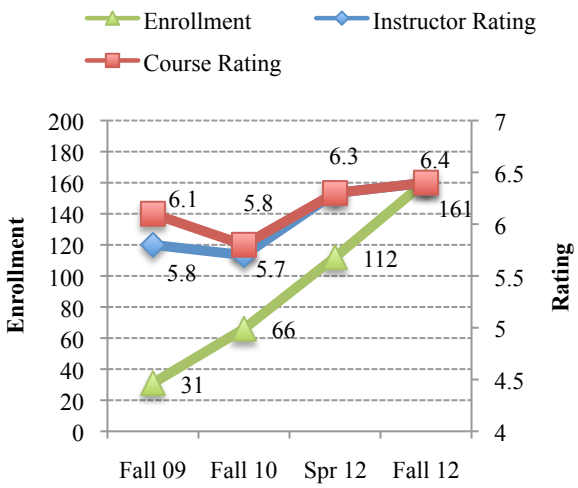

Figure 1. Course enrollment and instructor and course ratings (given anonymously by enrolled students, solicited by Eta Kappa Nu Engineering Honor Society each semester within Berkeley Engineering) of CS 169 Software Engineering. The first two offerings are without the SPOC and the last two are with the SPOC (see Section 5). The course continues to grow; there are 240 students in the Fall 2013 course.

2.5 Evaluations of the UC Berkeley Course

We have offered the course four times over the last four years. The first evaluation is students voting with their feet. Enrollments have grown with Moore's Law from 31 to 161, as Figure 1 shows. Note that the quantitative evaluation from students has increased

as well. Looking at the past 20 years' offerings of this course, we have set records for both the size of the class and the average numerical rating from the students of the class and its instructors.

We also polled past students to see what they thought of the material after they graduated and took jobs in industry. Figure 2 shows the survey results of Berkeley students from two earlier course offerings. Just 22 of the 47 respondents had graduated, and just 19 had done significant software projects. The figure shows the results of their 26 software projects.

We were surprised that Agile software development was so popular (68%) and that the cloud was such a popular platform (50%). Given that no language was used in more than 22% of the projects, our alumni must be using Agile in projects with languages other than Ruby. All the class teams had 4 or 5 students, which happily matches the average team size from the survey.

Figure 3 shows the alumni ranking of the topics in the course in terms of usefulness in their industrial projects, this time based on students from the Spring 2012 class, whose content is more up-to-date. Note that we divided the evaluations into the alumni who had graduated and were working in industry (on the left) and those still in school (right).

The majority alumni in industry agreed that the top 11 topics in the course were important in their jobs and the plurality agreed with the statement for all but 2 of the remaining 6 topics: pair programming and velocity. This result is understandable, since few organizations use pair programming and progress can be measured in other ways in industry than with velocity. Those who are still students didn't agree as strongly as those in industry about the importance of enhancing legacy code, unit testing, scrum team organization, JavaScript, and Rails itself. Based on the differing perspective in Figure 3, we recommend making sure to include past students working in the "real world" when requesting feedback how to evaluate and revise a course.

Another group worth asking was the non-technical customers of the student projects: 92% said that they were happy or thrilled with their apps, and 48% tried to hire the students to keep working on their projects. Our final evaluation is anecdotal comments from industrial colleagues about the course:

I'd be far more likely to prefer graduates of this program than any other I've seen.

—Brad Green, Engineering Manager, Google Inc.

A number of software engineers at C3 Energy consistently report that this … course enabled them to rapidly attain proficiency in SaaS development. I recommend this … course to anyone who wants to develop or improve their SaaS programming skills.

—Thomas M. Siebel, CEO, C3 Energy,
founder and former CEO, Siebel Systems

3. Lessons from Ebooks

Given that we thought that we had a successful approach, we believed the next logical step in making the ideas more widely available was to write a textbook that captured our approach.

We were both excited about exploring the potential of self-publishing electronic books, especially given that one of us has extensive publishing experience with print books. We saw many advantages that were particularly important for a book that would be closely related to software products.

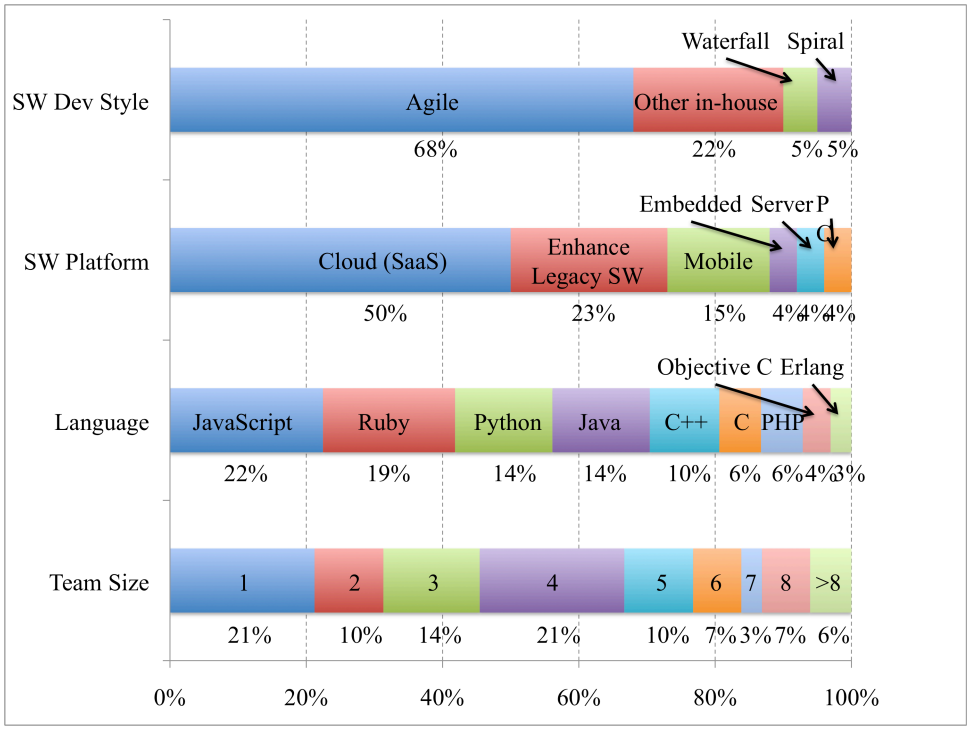

Figure 2. Survey results of software experience for former Berkeley students now in industry from two early offerings of the course.

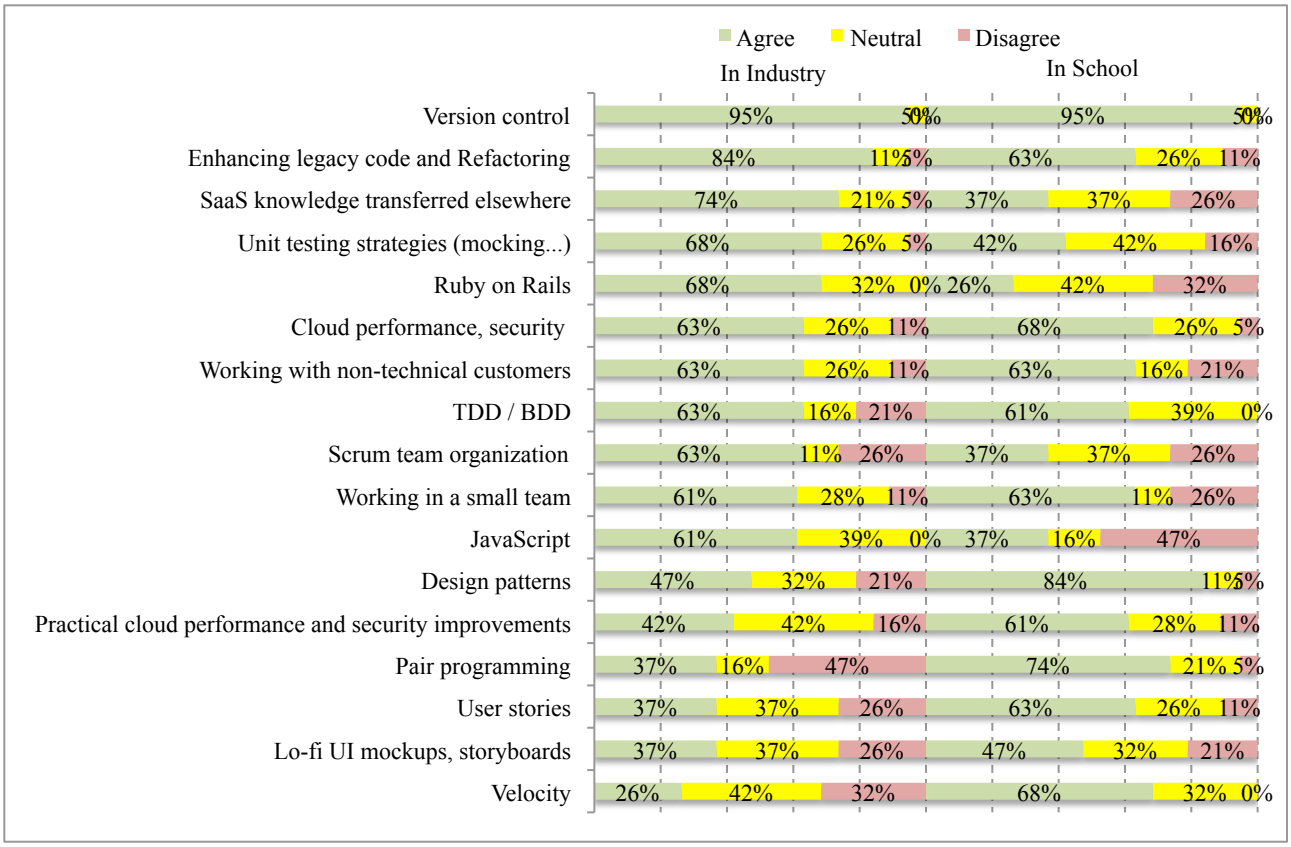

Figure 3. Ranked results from survey of CS169 alumni from Spring 2012 on whether course topics were important. On the left are responses from alumni in industry and on the right from those who have not yet graduated. The former much more highly value enhancing legacy code, reuse of SaaS knowledge, unit testing, Rails, scrum team organization, and JavaScript. The later more highly value design patterns, practical security and performance, pair programming, user stories, and velocity.

3.1 Iterative Development

With traditional print publishers, it can take nine months between when the authors are done and when the book is published. Ebooks allow us to release material within a week after we finish it. Thus, we offered alpha, beta, and 2nd beta versions of the book over 16 months to collect feedback and improve the material.

3.2 Frequent Editions

To amortize the costs of printing, a publisher will print thousands at a time, so editions are typically spaced at least two to three years apart. Not only will new releases of software tools appear that could make the text incorrect, in the Agile world new tools will appear would be extremely attractive to include. (We have added a new tool every time we have taught the class.) Ebooks enable new editions to be made as frequently as authors desire. We expect to need a new edition every year, and during our alpha and beta editions we pushed out updates every couple of months.

3.3 No Errata

Since an Ebook is electronic media, we can issue a new revision with typos fixed as frequently as we wish and readers will get their Ebooks automatically upgraded. This advantage is particularly important when the book is tied to software tools, as it is exceptionally frustrating if you do exactly what the book says and it doesn't work. So far, we have made about a dozen releases across all editions.

3.4 Low Cost

As there are no middlemen when you self-publish, we were able to keep the costs low ($10). This made the book affordable around the world, which proved to be important for the MOOC.

3.5 Print on Demand Turns Ebooks into Print Books

We were pleasantly surprised to see that Ebooks have been paired with print-on-demand (POD) publishing. Like Ebooks, there are no warehouses full of POD books that must be sold before authors can release a new edition. Thus, we were able to offer print books for readers who prefer them while keeping iterative development and frequent editions. POD books cost much more than Ebooks, but as self-publishers we can offer it at roughly a third the price of traditional print textbooks. We need to offer errata sheets for POD books, but that is not too heavy a price to pay to support those who prefer to read print books.

4. Lessons from MOOCs[6]

After we made a pact to write an electronic textbook, Daphne Koller and Andrew Ng approached us in October 2011 to teach

[6] This section is derived from Fox and Patterson [12] and Fox [13].

our course online via what we thought was Stanford University. Thus, we taught UC Berkeley's first MOOC in February 2012, and it was one of the first courses offered by Cousera, the company Koller and Ng founded. Our university has since decided to partner with EdX, so our courses are now called CS169.1x and CS169.2x, Software as a Service Parts I & II, on EdX. (We have two courses because we divided our course into two segments; see Section 4.5.)

As with any disruptive technology, there are bound to be some pitfalls along the way. How can instructors new to MOOCs successfully navigate teaching a MOOC? Below are tips from our experience in case others want to do a MOOC. All in all, it's way more work than "just" owning an on-campus course, but it's also tremendously rewarding.

4.1 Having A Rerun Plan Is Better Than Being Perfect

Leonardo da Vinci said, "Art is never finished, only abandoned." We found that while we could always find ways to improve our material, we could always revise our lecture recordings later—in Fall 2013 we are revising our MOOC lectures for the third time. We balanced our desire to perfect the material with the need to juggle all the other commitments most faculty must manage. Another perspective is that we needed feedback from MOOC students before we could improve it ourselves. Instead of obsessing about trying to get it right the first time, we focused on sustainability: Once we invested the enormous amount of work required to do a quality MOOC, we asked what resources will we need to re-offer the MOOC between refreshes of the material? We've managed to offer our MOOC two to three additional times between refreshes using World TAs (see the next section).

4.2 Consider Delegating

Most Berkeley campus courses use student discussion forums, and as conscientious instructors, we're used to checking the forums and posting answers to questions there frequently. But on-campus course forums tend to follow a regular rhythm as students work during the day, go to sleep (eventually), prepare for exams, or enjoy a short break following an exam or during a holiday. The cross-cultural, cross-time-zone reach of MOOCs obliterates this rhythm, and we found it too time-consuming to keep up with the forums. The challenge was exacerbated by the fact that most MOOCs don't have formal office hours or other means for students to get direct help, so the forums are even more critical to the student experience.

The first time we offered the course we recruited some of the strongest undergraduates from the previous campus offering of the course to serve as forum monitors. On subsequent offerings, we recruited volunteer "World TAs" from among the highest-scoring MOOC students, and retained an undergraduate working about 20 hours a week to organize the volunteers' efforts as well as serving as "Head TA." This system has worked well: the world TAs get some recognition, the course gets forum coverage by multilingual students spanning all the time zones (in our most recent offering, there was coverage nearly 24x7), and we get our lives back. We still check in every week or two with our Head TA to see how things are going, and often do 5-minute impromptu videos (Prof. Jennifer Widom at Stanford called them 'screenside chats') on topics in the news relevant to that week's course content.

4.3 "On The Internet, Nobody Knows You're A Dog"

The New Yorker magazine famously printed this caption in the early nineties to draw attention to the anonymity available on the Internet. Unfortunately, a small fraction of MOOC students take advantage of anonymity to engage in antisocial or antagonistic behavior on the forums, towards either their fellow students or the course staff. We found that these perpetrators were cowards hiding behind an anonymous throwaway email address. Up to a certain point we could instruct our World TAs to shut down destructive threads, but if the behavior persists, we recommend trying to have the students expelled from the course. We tried to not let their behavior sour the experience for the vast majority of students who are diligent and appreciative of our work!

4.4 Dry Run the Technology

With thousands of students, course technology has to work perfectly. We extended the EdX platform with sophisticated autograders for our programming assignments. Critical to our success was "dry running" new autograders and new assignments in our campus classroom to fix both logic bugs in the autograders and problems with the grading rubrics for new homeworks. We started the MOOC three weeks after the campus course to give to us time to repair assignments and autograders. Dry runs save a world of pain.

4.5 Divide to Conquer

Rather than create a single 12-week MOOC in one fell swoop, we first created a 6-week MOOC (CS169.1x), and offered it a few times. The next semester we recorded the second 6 weeks of the campus course to make CS169.2x, and then told the CS169.1x alumni that part 2 was available. Instead of one long marathon, we (and our families) were very glad we split the 12 weeks of MOOC across two offerings to give us time to recover.

4.6 Evaluate the Data

The large enrollments of MOOCs offer us new and unprecedented opportunities to improve our on-campus courses using inferential statistics techniques that just don't work at smaller scales, and so were previously available only to large-enrollment "high stakes" exams such as the GRE or SAT.

For example, *exploratory factor analysis* lets us identify questions that test comparable concepts, giving instructors a way to vary exam content [16]. *Item response theory* allows us to discover which questions are more difficult (in the statistical sense that higher-performing students are more likely to get them right) [17]. *A/B testing* gives us a controlled way to evaluate which approaches have better effects on learning outcomes, just as high-volume e-commerce sites evaluate which user experience results in more purchases. None of these techniques works on classroom-sized cohorts (say, 200 or fewer students), but we are applying all of them to our current MOOC.

Our sense at Berkeley is that MOOCs may well raise the bar for acceptable teaching on campus, as well as improve the recognition of good teaching, perhaps bringing the era of recycled PowerPoint slides finally to a close.

4.7 If It Hurts, Don't Do It

One criticism is that many aspects of traditional classes, such as small-group discussions and face-to-face time with instructors, do not work in the MOOC format.

This assertion is true, but it implicitly and incorrectly assumes that replicating the classroom experience is the proper goal for an online course. If that were an appropriate goal, then MOOCs would indeed fail to meet it. However, as educators, a better question for us to ask is this: What can be delivered effectively through this medium in a way that helps our on-campus students, and has the valuable side effect of helping the hundreds of thou-

sands who won't have the privilege of attending our universities in person?

For example, rather than asking whether automatic graders (which, by the way, have been around since at least 1960 [18]) can replace individual instructor attention, we can ask: When can they relieve teaching staff of drudgery, allowing scarce instructor time to focus on higher-value interactions such as tutoring and design reviews? Rather than worrying whether MOOC-based social networking will replace face-to-face peer interactions, we can ask and experimentally answer: Under what conditions and with what types of material do online communities help foster learning, and how can social networking technology help foster both online and in-person community building? And learning activities that don't appear to be "MOOCable"—discussion-based learning, open-ended design projects, and so on—can just be omitted from the MOOC but covered in the classroom setting, as we've done in our software engineering course, whose MOOC version lacks the on-campus course's open-ended design project.

Indeed, at universities on the quarter system, it's common to offer a two-quarter sequence in which the first quarter focuses on well-circumscribed assignments and the second quarter focuses on a design project, since a single quarter can't cover both. The first course clearly has value despite lacking a design project, and could be offered as a MOOC. By analogy, MOOCs that don't offer "the same" experience as a complete residential course also have value, and our job as educators is to make judgments about where that value lies and how to combine it with the other education modalities we offer our students. As a concrete example, our MOOC does not offer team projects or pair programming, which are important pieces of the Berkeley course. Nevertheless, many of our MOOC students reported that our course was better than anything available at the brick-and-mortar campuses to which they had access.

5. Lessons from SPOCs

Our and others' surveys of MOOC students have found that they are not like our campus students. Three-fourths live outside the North America (Table 3), but more importantly, roughly the same fraction are working full time (Table 1) and already have college degrees (Table 2).

Table 1. Primary Occupation

High school student	1%
Undergraduate student	8%
Graduate student	5%
Raising a family at home	1%
Full time job	70%
Part time job	6%
Unemployed	8%

Table 2. Highest level of education completed.

Less than high school degree	1%
High school degree or equivalent	8%
Some college but no degree	11%
Associate degree	3%
Baccalaureate degree	32%
Professional degree (JD, MD, ...)	11%
Graduate degree (MS, MA, PhD, ...)	35%

Thus, despite widespread fears of MOOCs undermining undergraduate education, thus far they are primarily a threat to continuing education programs.

MOOCs helped with our goals of educational technology transfer by dramatically expanding our classroom both numerically and geographically—10,000 students from 113 countries earned certificates from our MOOCs in 2012—but they have had less affect on conventional undergraduate courses, which was our original goal. The good news was that nearly 10% of the MOOC students said they were instructors, so that meant the MOOCs were helping us teach the teachers, in the hopes that they would incorporate our material into their courses.

5.1 Defining SPOCs[8]

It seemed that there must be more we could do to share all the technology we developed for the MOOC to make it easier for instructors to teach software engineering in the way we developed. For example, in a recent pilot program at San José State University in California, students in an analog circuits course used MIT-authored MOOC lectures and homework assignments created by Prof. Anant Agarwal. The students' in-classroom time was spent working on lab and design problems with local faculty and TAs.

The SJSU students in this *SPOC* (*Small, Private Online Course*) scored 5 percentage points higher on the first exam and 10 points on the second exam than the previous cohort that had used the traditional material. Even more strikingly, the proportion of students receiving credit for the course ("C" or better grade) increased from 59% to 91%. So educational quality arguably increased, and costs were lowered by helping students graduate more quickly, rather than by firing people. Productivity was enhanced because the on-campus instructors shifted their time from what they perceived as a lower-value activity—creating and delivering lectures on content that hasn't changed much—to the higher-value activity of working directly with students on the material. This model takes advantage of important MOOC features, including access to high-quality materials and rapid feedback to students via autograding, to maximize the leverage of the scarce resource—instructor time.

5.2 SPOCs at Berkeley

A key feature of our software engineering course is four different autograders for different types of software engineering assignments. These autograders were created by investing several hundred engineer-hours in repurposing tools used by professional programmers. Students not only get finer-grained feedback than they'd get from human TAs, who can spend at most a few minutes per assignment, but now have the opportunity to resubmit homeworks to improve on their previous score and increase mastery. We plan for future releases to give feedback on coding style and test completeness as well as simply code correctness.

A Figure 1 shows, the SPOC model has allowed us to increase the enrollment of the course nearly fourfold while yielding higher instructor and course ratings even though the fundamental material covered has changed very little.

5.3 SPOCs Beyond Berkeley

As part of the beta-testing program for the book, we recruited instructors with courses from four universities to try both the book and the MOOC in Spring 2013:

[8] This section is derived from Fox [13].

Table 3. The Top 50 countries of MOOC students of CS169.1x. The total number of countries was 113.

Rank	*Percent*	*Country*	*Running Total*	*Rank*	*Percent*	*Country*	*Running Total*
1	19.7%	United States	19.7%	26	0.6%	Belarus	80.7%
2	10.4%	Spain	30.1%	27	0.6%	Egypt	81.3%
3	7.0%	India	37.1%	28	0.6%	Netherlands	81.9%
4	5.7%	Russian Federation	42.8%	29	0.6%	Sweden	82.5%
5	5.3%	United Kingdom	48.1%	30	0.6%	Algeria	83.1%
6	3.8%	Brazil	51.9%	31	0.6%	Costa Rica	83.7%
7	3.3%	Canada	55.2%	32	0.6%	Czech Republic	84.3%
8	2.6%	Ukraine	57.8%	33	0.6%	Philippines	84.9%
9	2.3%	Germany	60.1%	34	0.5%	Bulgaria	85.4%
10	2.2%	Australia	62.3%	35	0.5%	Indonesia	85.9%
11	2.1%	Poland	64.4%	36	0.5%	Peru	86.4%
12	1.9%	France	66.3%	37	0.5%	Austria	86.9%
13	1.8%	Italy	68.1%	38	0.5%	Ghana	87.4%
14	1.5%	Portugal	69.6%	39	0.5%	Ireland	87.9%
15	1.3%	Pakistan	70.9%	40	0.5%	Malaysia	88.4%
16	1.2%	Argentina	72.1%	41	0.5%	Singapore	88.9%
17	1.2%	Greece	73.3%	42	0.4%	Bolivia	89.3%
18	1.0%	Hungary	74.3%	43	0.4%	Denmark	89.7%
19	1.0%	Mexico	75.3%	44	0.4%	Israel	90.1%
20	1.0%	Romania	76.3%	45	0.4%	Serbia	90.5%
21	0.9%	Colombia	77.2%	46	0.4%	Turkey	90.9%
22	0.8%	Nigeria	78.0%	47	0.3%	Chile	91.2%
23	0.7%	Switzerland	78.7%	48	0.3%	Ethiopia	91.5%
24	0.7%	Belgium	79.4%	49	0.3%	Finland	91.8%
25	0.7%	South Africa	80.1%	50	0.3%	Kenya	92.1%

- *Binghamton University*: 14-week elective software engineering course with team projects taken by sophomores and juniors.
- *Hawaii Pacific University*: 15-week required systems analysis/software engineering course for seniors with individual student projects.
- *University of Colorado, Colorado Springs*: 16-week required software engineering course with team projects for juniors and seniors. Some of MOOC lectures were also used to supplement a graduate class in Software Engineering.
- *University of North Carolina, Charlotte*: 15-week required Software Engineering course with group projects for sophomores and juniors.

These faculty were either unhappy with the current textbooks or more interested in Agile than Plan-and-Document methodologies, as well as being interested in using materials that were readily available to reduce their workloads. All faculty watched the MOOC lectures to prepare for the course, and three of the four used the exams. Two used the autograded assignments in their courses; one had students watch the MOOC videos in addition to lectures, and one "flipped the classroom," where students are watch the videos on their own instead of their instructor's lectures, and the classroom becomes more like a discussion section.

Here were some of the problems:

- Some students' computers were too slow to run the VM.
- Some students were not familiar with Linux, which added to their learning curve.
- Since thousands did the assignments, it was inevitable that solutions would be easily available on the Internet.
- Autograders checked for correct "output," but did not check code style. Until we can get autograders to evaluate quality metrics, as mentioned above, it would still be desirable for humans to review the students' code as well.
- Because of some of the logistical problems (with the autograders, the programming environment, and so on) some students took this as an excuse the cut back their efforts.

Here is what worked well:

- Auto-graders took the grading burden off the staff, while simultaneously reinforcing the notion of test-driven development.
- Video lectures were a highly efficient way to convey information. They were dense with information, but students could pause and review at any point.
- Students are excited being introduced to the latest technology (Rails) and leading edge development methodologies (Agile).
- The course provided the better students challenges they were not getting in their other classes.
- Students are impressed that they're getting "world-class" instruction (via the video lectures) and being challenged by the same curriculum given at a top-tier computer science program.
- Several students got jobs from material learned in this class.

While the start-up logistics were challenging, all were interested in participating again in Fall 2013, and we are working to address the shortcomings that they uncovered in the next course offering.

One improvement is to have the SPOC students participate in the MOOC forum so that they could benefit from talking to students at other schools. The beta-test faculty observed that many students were having the same issues, particularly on the homework assignments. Having a larger community with whom to discuss challenges and issues would help, especially when they were first beginning with new languages and tools. With the MOOC system, they could have a much larger range of responses and perspectives than with just their small class group. A larger discussion group might also give them a different perspective on

software engineering. For example, students with industrial experience in one SPOC were appalled when hearing negative comments from their classmates about writing tests, but the issue didn't arise until the student presentations at the end of the semester. The MOOC Forum would have likely addressed the topic earlier in the course. The MOOC forum could also help by leveraging the World TAs to answer questions. Having experienced TAs is especially helpful given the new language, framework, and tools, and such TAs can be hard to come by on any campus.

6. Conclusion

Cloud computing and the shift in the software industry towards software as a service has led to highly-productive tools and techniques that are a much better match to the classroom than earlier software development methods. That is, not only has the future of software been revolutionized, it has changed in a way that makes it *easier* to teach.

UC Berkeley's revised Software Engineering course leverages this productivity to allow students to both enhance a legacy application and to develop a new app that matches requirements of non-technical customers. By experiencing whole software life cycle repeatedly within a single college course, students actually use the skills that industry has long encouraged and learn to appreciate them. We believe it demonstrates one way to address the many challenges of teaching software engineering.

This revision pleases many stakeholders:

- Faculty like it because students actually use what they hear in lecture, even after graduation, and they experience how big CS ideas genuinely improve productivity.
- Students like it because they get the pride of accomplishment in shipping code that works and is used by people other than their instructors, plus they get experience that can help land internships or jobs.
- Colleagues in industry like it because it addresses several of their concerns.

Thus, the course is now heartening to faculty, popular with students, and praised by industry. To transfer this educational technology to other institutions, we tried Ebooks and MOOCs.

Ebooks are a great match to a software course, as they simplify making corrections and bringing out new editions to keep pace with the rapidly evolving software tools. They also are a boon to authors in that they allow us to do extensive class testing and let us publish a book much more quickly. They also benefit readers since Ebooks encourage self-publishing, which can lower prices. Most importantly, our MOOC would likely have been too challenging for most students *and* instructors if not for the Ebook.

MOOCs represent a new technology opportunity whose potential pedagogical impact needs to be researched. We argue that MOOCs themselves can yield valuable information because of their scale, and that MOOC materials can be used in a blended setting called SPOC or Small Private Online Course to supplement the classroom experience.

Some have speculated that MOOCs will become the 21st century textbook. Based on our experience, we think the new paradigm will is more likely the combination of Ebooks and SPOCs, as they are complimentary and synergistic. We believe you can just pack more detailed and precise information in a 400-page Ebook than you can in 12-weeks of lecture.

Both MOOCs and SPOCs are two design points in a wider space in which experiments are possible. To be sure, many bad experiments will be tried—some are probably already underway—and many worthy experiments will fail or have a different outcome than desired. But if failed experiments were an obstacle to doing world-changing research, we academics would probably choose a different job.

Acknowledgments

In addition to all the Berkeley and MOOC students, we wish to thank the faculty and students of our beta program: Richard Ilson (UNCC), Samuel Joseph (HPU), Kristen Walcott-Justice (UCCS), and Rose Williams (Binghamton).

References

[1] Waxer, C. "Software Engineer: 2012's Top Job," *InformationWeek*, May 15, 2012.

[2] Pressman, R. *Software Engineering: A Practitioner's Approach*, 7th Edition. McGraw Hill, 2010.

[3] Fox, A., & Patterson, D. "Crossing the software education chasm." *Communications of the ACM*, 55(5), 44-49, 2012.

[4] Fox, A., & Patterson, D. "Is the New Software Engineering Curriculum Agile?" *IEEE Software* 30.5 (September/October 2013): 101-104.

[5] Joint Task Force on Computing Curricula, "Computer Science Curricula 2013, Ironman Draft (version 1.0)," ACM/IEEE CS, Feb. 2013, http://ai.stanford.edu/users/sahami/CS2013/.

[6] Estler, H.-C., et al., "Agile vs. Structured Distributed Software Development: A Case Study," *Proc. 7th Int'l Conf. Global Software Eng.*, IEEE 2012, pp. 11–20.

[7] Sommerville, I. *Software Engineering*, 9th ed., Addison-Wesley, 2010.

[8] Feng, J. & T. Sedano. "Comparing Extreme Programming and Waterfall Project Results" *Conference on Software Engineering Education and Training 201*1 (2011).

[9] Stella, L., S. Jarzabek, & B. Wadhwa, "A comparative study of maintainability of web applications on J2EE, .NET and Ruby on Rails," *WSE 2008. 10th International Symposium on Web Site Evolution*, pp.93-99, 3-4 Oct. 2008.

[10] http://en.wikipedia.org/wiki/Cucumber(software

[11] McCabe, T. "A complexity measure." *Software Engineering, IEEE Transactions on* 4 (1976): 308-320.

[12] Fitzpatrick, J. "Applying the ABC metric to C, C++, and Java." *C++ Report*, June 1997.

[13] Patterson, D. & J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th Edition, Morgan Kaufmann Publishers, 2014.

[14] Fox, A., & Patterson, D. "What We've Learned from Teaching MOOCs." https://www.edx.org/blog, May 8, 2013.

[15] Fox, A. "From MOOCs to SPOCs." *Communications of the ACM*, to appear.

[16] Lawley, D., Estimation of factor loadings by the method of maximum likelihood. *Proc. Royal Soc. Edinburgh*, 60A, 1940.

[17] Lord, F.M. *Applications of item response theory to practical testing problems*. Mahwah, NJ: Erlbaum, 1980.

[18] Hollingsworth, J. Automatic graders for programming classes. *Communications of the ACM* 3(10), Oct. 1960.