Towards a Supercollaborative Software Engineering MOOC

William Billingsley
NICTA
GPO Box 2434, Brisbane
Queensland 4001, Australia
william.billingsley@nicta.com.au

Jim R. H. Steel
The University of Queensland
Building 78, St Lucia
Queensland 4072, Australia
j.steel@uq.edu.au

ABSTRACT
Recently there has been rapid growth in the number of online courses and venues through which students can learn introductory computer programming. As software engineering education becomes more prevalent online, online education will need to address how to give students the skills and experience at programming collaboratively on realistic projects. In this paper, we analyse factors affecting how a supercollaborative on-campus software studio course could be adapted as a project-led supercollaborative MOOC.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education*

General Terms
Design, Human Factors.

Keywords
Software Engineering, Massively Open Online Course, Studio Course, Continuous Integration.

1. INTRODUCTION
Recently, there has been rapid growth in the number of online MOOCs (massively open online courses) and venues teaching introductory computer science and programming. Coursera, edX, and Udacity each offer multiple such courses. In 2012, 150,349 students enrolled in CS50x from Harvard University on edX, with 1,388 receiving a certificate of completion [1]. Khan Academy, Codecademy, Treehouse, and other sites also offer students support for online study in programming.

These courses predominantly teach students to program on their own. BerkeleyX's Software as a Service course [2] includes collaborative topics and encourages pair programming, but the scale of collaboration students undertake is small. As software engineering education becomes more prevalent online, we suggest that online education will need to address how to give students the skills and experience of programming collaboratively on realistic software projects. This has been an active topic of research for on-campus courses for more than twenty years [3, 4, 5], but is an area that online education is largely yet to address.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'14, May 31 – June 7, 2014, Hyderabad, India.
Copyright 2014 ACM 978-1-4503-2768-8/14/05 …$15.00.

In previous work [6, 7] we described an on-campus course centred around a software project that has extensive inter- as well as intra-group collaboration. Students work in small groups to develop a feature for a program, but must coordinate and integrate with many other groups also contributing features. In 2011 and 2012, we had approximately 70 students working on a single codebase; in 2013 this rose to over 140 students. For the purpose of this paper, we refer to this kind of project, in which students collaborate in groups which in turn collaborate with other groups, as a *supercollaborative* project.

We are adapting our course to offer it as a MOOC. Online students would access videos, tutorials, exercises, and can take part in supercollaborative projects. On-campus students would have the additional resources of lab studio time, and interactive workshops and demonstrations by the lecturers in class.

In this paper, we discuss considerations that impact on the course's viability online, and how the design will need to change. Until the MOOC runs we will be unable to fully evaluate the outcome, but we present this paper so that the analysis and issues involved may be shared with the software engineering education community.

2. ON-CAMPUS COURSE SUMMARY
The course was developed in 2011, and for the first two iterations was predominantly taken by second-year students in the software engineering and information technology programs. In this version of the course, students collaborated on developing features for a fork of the Robocode open source project.

A restructuring of the school's degree programs caused the class size to more than double in 2013, with the addition of many multimedia design and information systems students. In this iteration of the course, we changed the project to give students a much smaller bespoke starting code base to build from. The project we chose was to build a multiplayer games arcade.

Code collaboration takes place on GitHub, supported by continuous integration servers that build each commit pushed to the master branch of the repository. Automated builds, test coverage, and automated software metrics give the class ready access to information about the health of the project.

Alongside the project, a weekly lecture series covers topics on distributed version control, automated builds, continuous integration, debugging, testing, design patterns and other topics relevant to their collaborative project. An unstructured weekly studio session is also provided, giving teams a weekly timeslot when they have access to in-person support in their project work from tutors, the lecturers, and each other.

In 2013, 172 students enrolled at the start of semester, with 146 still enrolled in the final week. Over the semester, the class made 4,883 commits and grew the project to 67,900 lines of code. The students developed included many games as well as cross-cutting

features such as a common overlay, achievements, networked multiplayer support, and an event recording/replay facility.

3. IMPACT OF CLASS SIZE AND ATTRITION ON VIABILITY OF SUPERCOLLABORATION

While an on-campus course of 172 can place all the students on the same project, a MOOC that could potentially have orders of magnitude more students will need to subdivide the class into many project cohorts.

MOOCs typically have also a much higher rate of attrition than on-campus courses. The quoted attrition rates can be 90% or more. For example, of 150,349 students registering for CS50x only 1,388 (0.9%) received a certificate of completion [1]. For MITx's first course, 6.001x, of almost 155,000 registering only 7,157 (4.6%) passed the course [8], and Philip Zelikow's Coursera course on the history of the modern world passed nearly 5,000 (10.6%) of nearly 47,000 students [9].

Most of this attrition occurs early, and for many MOOCs a high attrition rate might not be problematic as students can gain value from a course without completing it. For CS50x, only 10,905 (7%) students submitted the first problem set, 10,137 (7%) indicated they intended to complete all the coursework, and only 3,381 (2.3%) indicated they took the course because of the prospect of the course certificate [1]. For 6.001x, 23,000 students (14.8%) submitted the first problem set, and 9,000 (5.8%) passed the midterm quiz.

However, attrition could have a much more adverse effect in our course. In courses that are taken individually, if a student drops out they are the only one affected. In a team course, students dropping out could cause the team to become unviable, leaving their teammates "stranded". In a supercollaborative course, if too many teams on a project become unviable, the project may become unviable or ineffective, affecting the other teams. There is a question, then, of whether the high attrition rate of a MOOC would cause a supercollaborative course to become unviable. The projects must be manageable when student numbers are at their peak, but must remain viable and pedagogically effective as participant numbers fall.

If the viability of the projects can be ensured, then the attrition within teams can be mitigated and potentially used to pedagogical advantage. Reallocating stranded members to other teams tests both the arriving member, in comprehension of a new system, and the team, in providing transparency and documentation of the group's product and processes. In the on-campus course, students undertake a single long sprint, allowing them to encounter all of the course content before a sprint cycle finished. But in an online course, shorter sprints may be preferable, to better reflect real practice and to provide a natural point at which to reallocate students between teams.

3.1 Minimum Pedagogically Effective Size

When we designed the on-campus course, one of our design principles was that students should be faced with problems for which the most effective solution is to use the techniques that we are teaching.

This was one of the reasons for putting multiple groups on the same codebase. On small projects, with only intra-group collaboration, it is feasible for students to coordinate their activity just by talking to each other, without using the processes, discipline, and tools that the course teaches. It is when someone you do not talk to frequently starts modifying your code in unexpected ways that you discover the value of good tests, version control, and an issue tracker. By requiring inter-group collaboration, we scaled the project far beyond what conversation alone could support, and forced students to encounter the problems that the course teaching addresses rather than circumvent them. We dubbed this the "feel the pain" pedagogy.

In a MOOC, students are generally remote from each other, although there will be some groups of students who are colocated. Students can join a MOOC from all around the world, but some students who already know each other might decide to sign up together, and local study groups are often formed.

So long as a project does not consist only of colocated teams, it would be difficult for students to coordinate through conversation alone. Remote communication would be a barrier. This suggests that in a MOOC environment, the minimum size of a project for it to be pedagogically effective could be smaller than in an on-campus environment. In the three iterations of the on-campus course so far, we have always had at least nineteen teams collaborating on a single codebase. Online, a project (a single codebase) could potentially remain pedagogically effective with far fewer.

3.2 Maximum Manageable Size

If attrition is to occur throughout the course, this would suggest that the projects would need to be reasonably large at the beginning so that they do not shrink too small by the end. However, even at its peak size, each project must be manageable.

There are two aspects of the project that we consider may become unmanageable: the code base, and initial feature selection.

With more students and more teams modifying the code, we would expect the code base to change more rapidly. This could make it difficult for some students to keep up with a fast-moving target. However, on campus we have found that the code changes fastest at the end of the project, as teams rush to finish their work before the due date. Figure 1 shows the number of commits per week in the on-campus course in 2013. This end-of-project rush does cause problems for students and is something that we are seeking to reduce. But it is worth noting that in a MOOC it would occur when attrition has already taken place and projects are at their smallest.

Feature selection, where groups decide what they would like to work on, occurs earlier in the course. While groups may implement more than one feature, typically they have one feature that is their main focus, and usually it is the one they first selected.

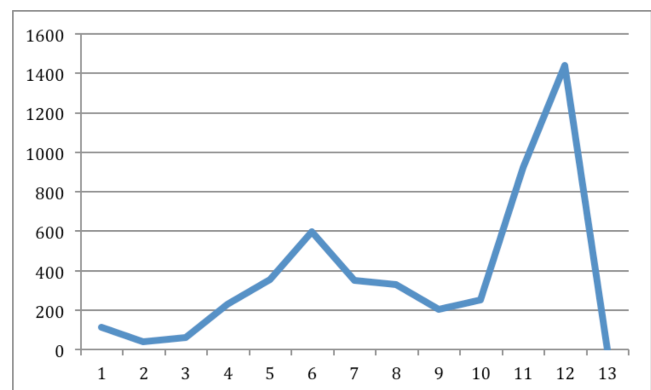

Figure 1: Commits per week in the on-campus course in 2013.

This feature selection drives aspects of both the collaboration and the design. As teams try to ensure that their own work is coherent (so they have a clear vision of what they are trying to achieve) and has loose coupling with other teams (so they do not need to wait on each other's changes), the teams' choices of features can be seen as a quickly established high-level design for the system.

In 2013, thirty-one teams collaborated on the project. Fifteen of those teams worked on infrastructure features and APIs that could be consumed by other teams, and nineteen teams developed games. (Three did both.) While nineteen (in 2012) and thirty-one (in 2013) teams succeeded in choosing well-fitting features with a reasonable separation of concerns, that number probably does not scale up indefinitely. At some project size, there would be too many teams in the system for each student to have a reasonable grasp of what the other teams are choosing to do.

In our view, then, it is feature selection rather than code velocity that is likely to impose an upper limit on the viable project size.

3.3 Individual Work as well as Group Work

One strategy for minimizing attrition in the projects would be to begin the project work after much of the course attrition has taken place. This also fits the practical needs of the project structure.

In Figure 1, there are comparatively few commits until week 4. Before students can begin making significant changes to the code, they need to obtain it, understand some part of it, and have a reasonable idea of a change they would like to make. Thus the first part of the course requires learning version control (to get the code), build systems (to get the code working), and ways of exploring and understanding the existing code. This has to happen before collaboration can start taking place. We ask students to form their groups by week 3 of the course, and to start establishing what feature they would like to work on in week 4.

CS50x is the MOOC with the lowest nominal completion rate that we have found, so makes for a suitable pessimistic scenario to test against. In CS50x, 3,292 students submitted the problem set in week 3, and 5,259 students completed the subsequent quiz [1]. If we take the higher of those two numbers (rather than course registrations) as our base, then the attrition rate from there on in is 73.6%, rather than 99.1%. This is still very high, but in this scenario supercollaboration would be viable.

Figures 2 and 3 show the mean results of twenty runs of a computer simulation in which students are placed in teams of ten, with fifty teams per project. The simulation repeatedly removes a student from a random team according to the attrition rate. The chart shows the survivability of teams, and the survivability of projects, where 150,000 students are placed into groups and then undergo 99% attrition, and where 5,250 students are placed into groups and undergo 74% attrition. In the former case, half the surviving teams are left with only a single team member, and half the projects are left with only a single team. However, in the latter scenario, the median team size is three and all of the projects in every run had at least fourteen active teams remaining.

The results here are not intended to be predictive. So far we know very little about the factors that affect MOOC attrition rates, and it is unlikely that students would leave groups randomly, but it suggests that even under a reasonably pessimistic scenario, supercollaboration is not obviously doomed to collapse. A lead time of individual work before the project begins can bring the attrition rate during the project below a rate where it is viable.

Automated quizzes and exercises, which are typical to most MOOCs, would also have two additional benefits for the course.

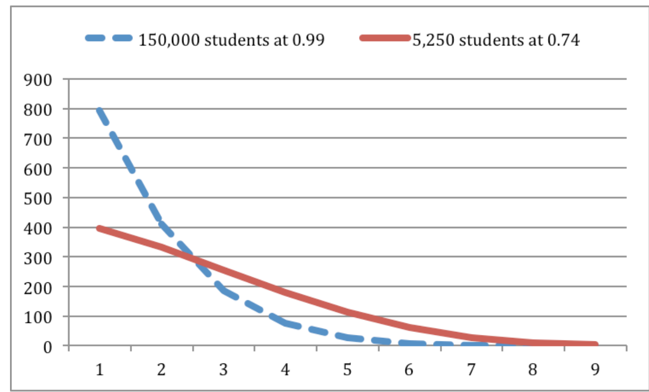

Figure 2: The number of teams with at least n students after attrition. Mean of twenty computer simulations.

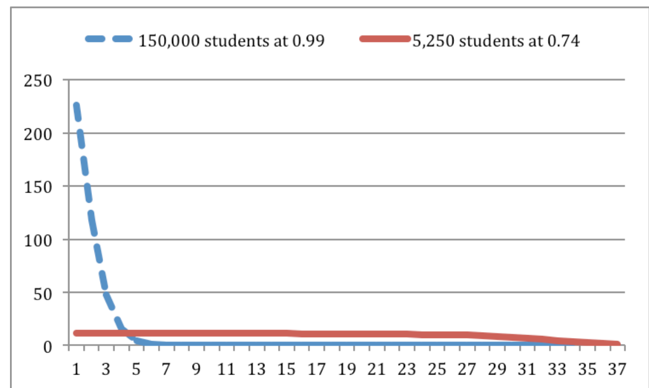

Figure 3: The number of projects with at least n non-empty teams after attrition. Mean of twenty computer simulations.

Firstly, by exposing students' activity (though not their marks) in individual exercises, the course could encourage active students to cluster together into teams. This would provide some additional defence against attrition, as the harm occurs when an active student finds that his or her collaborators are inactive.

Secondly, it would improve the course's support for trying out a technique in a tame environment, before applying it to the project. The project compels students to apply some concepts frequently, such as merging changes, and produces immediately visible outcomes for these. However, on-campus we observe that other topics (e.g. refactoring) have longer feedback loops. The teams consuming a code change might not be ready to work with it immediately, and students do not always view the static analysis reports. Automated exercises would ensure students can test their understanding of these topics with immediate feedback.

4. SELECTIVE PARTICIPATION AND HETEROGENEITY

In a traditional on-campus course, the knowledge, background, and interests of the students are heavily constrained. We know that students have taken the prerequisite subjects, and that they have not taken the subsequent subjects for which this is a prerequisite. We know that, as university students, they will have a broad focus across most of the topics we require for the exam, because they need to pass their courses in order to obtain their degrees. In a MOOC, this is not the case. Some students may already have degrees, while others may still be in high school.

In 2013, our on-campus cohort became somewhat more varied as multimedia design and information systems students came into the

course alongside the software engineers. To cater for this, we changed the project from improving a pre-existing product (Robocode) to creating a new system from a very small base. As the base was small, it was comparatively unconstrained, and teams were able to specialise. Some teams took it upon themselves to define the design language of the arcade that teams should fit into. Other teams built games. Others built infrastructure that is used entirely behind the scenes. In terms of the balance of teams, we regard this as a successful change.

For a MOOC cohort that is even more varied, the course would need to be more flexible still. Some students might only be interested in particular topics within the course, and have no time for the project. Others could be well versed in the material, but perhaps interested in taking part in a project with students in order to try taking on a more senior role on the team than they have in their workplace. In our design, the supercollaborative project would be the largest component of the course. Teaching content would comprise topic-based sequences of videos and exercises, with each topic sequence able to stand alone, but also forming part of a consistent narrative within the course. The aim would be to ensure that new topic sequences could be added as needed, without the course becoming disjoint.

In a software engineering course like this, there are also many voices other than the course staff that would be of interest to students. There are many professional materials, videos, presentations, and an increasing number of online playgrounds and tutorials available. In previous work, we proposed courses where smart exercises were supported by a dynamic ecosystem of socially discovered, written, and added content [9]. The goal of dynamically evolving courses is still far off, but we can begin by including interviews with practitioners from companies and open source projects, and other materials by people outside the course.

5. PEER MARKING OF GROUPS

In the 2013 course, we introduced a critique at week 8. Groups presented their progress, and sought feedback from their colleagues. Students were allocated five groups to mark and critique by filling in a simple survey. The textual feedback in each critique was then made available to the group being criticised, who would in turn mark the critique on whether it was objective, specific, actionable, constructive, and useful. In total 725 critiques were completed, and 3,075 reviews of critiques.

The only staff intervention we were called to undertake in the critique marking (apart from writing the software to support it) was handling a small number of requests to fix student errors – particularly where students had submitted a critique for one team that they intended to be for another. However, as each group received more than twenty critiques, even if uncorrected these errors would have had negligible impact on the marking.

In our on-campus course, teams' final project outcomes are marked by the course staff. Within the teams, students use a peer grading system to divide their project mark between them according to their contribution. However, it appears it would be viable to replace the staff marking with a sequence of peer critiques, that might also encourage students not to leave so much work for the final week's rush.

6. CONCLUSION

Supercollaboration in the classroom is comparatively new. As far as we know, our course is still the only software engineering course to place so many teams of students on a single project. Supercollaboration in the open online classroom, then, is always likely to be a step into the unknown. However, our analysis suggests that it is feasible, and we are in the process of producing such a course. The key aspects appear to be keeping the elements of the course coherent but independent, and ensuring that project groups only form after the initial high student attrition has occurred, when there is data to identify and group active students.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the assistance of our course tutors, Jackson Gatenby and Phillip Drew. as well as Jörn Guy Süß, with whom we co-designed the first iteration of the course. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

8. REFERENCES

[1] Malan, D. 2013. *Data, data, data (from CS50x)* http://harvardx.harvard.edu/blog/data-data-data-cs50x Accessed 23 Oct 2013.

[2] Fox, A. and Patterson, D. 2013. *CS169.2x Engineering Software as a Service: An agile approach using cloud computing - MOOC*. http://beta.saasbook.info/courses Accessed 24 February 2014

[3] Tomayko, J.E. 1991. Teaching software development in a studio environment. *Proceedings of the twenty-second SIGCSE tehcnical symposium on computer science education*. 300-303

[4] Docherty, M., Sutton, P., Brereton, M., and Kaplan, S. 2001. An innovative design and studio-based CS degree. *ACM SIGCSE Bulletin*. 33, 1, 233-237. ACM

[5] Hundhausen, C.D., Narayanan, N.H., and Crosby, M.E. 2008. Exploring studio-based instructional models for computing education. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education* (SIGCSE '08). ACM, New York, NY, USA, 392-396.

[6] Billingsley, W. and Steel, J. 2013. A comparison of two iterations of a software studio course based on continuous integration. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (Canterbury, UK, July 01 - 03, 2013). ACM, New York, NY, 213-218.

[7] Süß, J. G., and Billingsley, W. 2012. Using continuous integration of code and content to teach software engineering with limited resources. In *34th International Conference on Software Engineering* (Zurich, Switzerland, June 02 - 09, 2012). 1175-1184

[8] Hardesty, L. 2012. Lessons learned from MITx's prototype course. *MIT News*, 16 July 2012. http://web.mit.edu/newsoffice/2012/mitx-edx-first-course-recap-0716.html Accessed 23 Oct 2013.

[9] Anderson, N. 2013. U-Va. MOOC finds high attrition, high satisfaction. *The Washington Post*, 13 May 2013. http://articles.washingtonpost.com/2013-05-13/local/39241038_1_coursera-massive-open-online-courses-moocs Accessed 23 Oct 2013

[10] Billingsley, W. 2008. *The intelligent book: technologies for intelligent and adaptive textbooks, focussing on discrete mathematics*. Technical report UCAM-CL-TR-719, University of Cambridge