A Comparison of Two Iterations of a Software Studio Course Based on Continuous Integration

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

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

ABSTRACT
In previous work we introduced a software studio course in which seventy students used continuous integration practices to collaborate on a common legacy code base. This enabled students to experience the issues of realistically sized software projects, and learn and apply appropriate techniques to overcome them, in a course without significant extra staffing. Although the course was broadly successful in its goals, it received a mixed response from students, and our paper noted several issues to overcome. This paper considers experimental changes to the course in light of our previous findings, and additional data from the official student surveys. Two iterations of the course and their respective results are compared. Whereas our previous paper addressed the feasibility of such a course, this paper considers how the student experience can be improved. The paper also considers how such a course can be adapted for more heterogeneous cohorts, such as the introduction of an unknown number of design and database students, or the introduction of online students.

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

General Terms
Design, Experimentation, Human Factors.

Keywords
Continuous Integration, Software Engineering, Studio Course, Experience Report

1. INTRODUCTION
In 2011, (author and collaborator) designed a software engineering studio course around continuous integration, in which nineteen groups of three or four students implemented features on an existing piece of software, but with all nineteen groups committing to the same code base [1]. The motivation for designing the course this way was that many of the difficulties that software engineering practices are designed to mitigate do not appear on the small greenfield group projects that students traditionally would undertake in their software development courses. On small “toy” projects, edited only by a small group that sees each other every day, it is often possible for students to produce code that works without version control practices, code visualisation tools, good API design, testing, continuous integration, and ticket management. On a project with 60,000 existing lines of code and approximately 70 other committers, it is very much harder and the tools and practices become necessary to succeed. The development difficulties and the students’ need to overcome them create a series of teachable moments: occasions when students are predisposed to learn the material at hand [2]. The lecture content is delivered just-in-time to address issues the class should be facing in their project at a given stage. This can be thought of as a “feel the pain” pedagogy – by experiencing the pain of the problems that arise, students gain a greater understanding of why the techniques to mitigate those problems are important.

From a practical perspective, continuous integration also makes the class feasible for the teaching staff. The same tools that allow the class to monitor what is happening in the collective code-base, whether the tests are failing and what has been changed, also allow the teaching staff to monitor the class’s collective progress reducing the need to meet regularly with each small group. This allows a studio course to run with only the level of staffing and tutor support of a traditional lecture-plus-tutorial course.

Our previous paper addressed the fundamental principles of the course, and demonstrated that a course where all students commit to the same code-base and interact with each other’s code is achievable with limited resources, and that students do learn the software engineering techniques that allow them to overcome the problems of larger-scale software development. The collaboration systems helped us to identify where students were encountering and overcoming the relevant issues, and a class survey in the final week confirmed that students had gained a greater appreciation of continuous integration, class design, and unit testing.

However, our original paper noted some significant problems in the course design that were yet to be overcome. The official course feedback survey results (unavailable at the time of our original paper) also showed that the course had a mixed reception with students, and the student-delivered technical demonstrations were particularly heavily criticised.

In 2012, the second iteration of the course ran, giving us the opportunity to make changes and compare the outcomes from the two versions of the course. The motivation for and results of these changes are discussed. Whereas our previous paper addressed the feasibility of the course, this paper considers how the student experience can be improved. The paper also considers how such a course can be adapted for more heterogeneous cohorts, such as the introduction of an unknown number of design and database students, or the introduction of online students.
2. RELATED WORK

Many universities have employed studio courses to bridge the gap between student and professional practice. Carnegie Mellon University’s studio course is a sixteen-month program that forms 40% of a Masters degree, in which groups of 3 to 7 developers work on projects for paying customers [3]. Taylor University runs a studio program for undergraduates, with a typical enrolment of only eight students per semester [4]. Rice University designed a course [5] in which students worked on an established in-house open source project, but each group of six students required a senior developer from the project to be employed as a “manager”.

The use of studio mode for delivering undergraduate material in software engineering and related disciplines has also been adopted in a number of universities around the world in recent years [6, 7]. The use of studio mode has shown benefits not only in terms of student learning outcomes [7] but also in student self-efficacy and in peer learning [8]. {Author’s institution}, at which the course discussed in this paper was delivered, has long incorporated a studio stream in its design program [9], and is now incorporating this into other computing-related programs.

A study of recent IT graduates in Australia [10] reinforced the importance of exposing students to industrial practices during their studies. Other research has investigated the impact of techniques such as Test-Driven Development (TDD) and continuous integration in students’ group assignment work [11]. In 2012, Buffardi and Edwards [12] found that students’ adherence to desirable development behaviours was related to how helpful they found them (and hence, we would argue, to the problems resulting from not following those behaviours).

The unique aspect of our course is requiring a large number of undergraduates to cooperate developing a shared existing code base, using continuous integration to maintain the cohesiveness of students’ work and centrally monitor their progress.

3. COURSE SUMMARY

The course is predominantly taken by second-year students in the Software Engineering (in which it is a compulsory course) and Information Technology programs. The course prerequisites ensure that students have a basic understanding of programming, and the Java language. A follow-on course asks students groups to work on projects with industry representatives acting as clients.

The principal component of the course involves development of features for an existing code base. In 2011 and 2012 Robocode – a game in which students write algorithms to drive virtual tanks in battle against each other – was used as the code base. An initial individual task asks students to write a robot (to familiarise themselves with the product), but the group development task is to change the game itself. Since all students are working on the same code-base within a central repository, they must use software engineering disciplines to ensure that the quality of the code-base is maintained. As their features will require overlapping functionality, students must work across as well as within their groups to ensure that features interoperate appropriately, and make proper use of modularity techniques.

Course material introduces students to version control, automated build systems, test-driven development, continuous integration, software comprehension techniques, design patterns, models, refactorings, and other tools and techniques that are needed to manage their development.

4. EXPERIMENTAL CHANGES

Although the course’s first iteration was broadly successful in its goals, it received a mixed response from students, and our paper noted several issues to overcome. The changes made in response are described here, together with some additional changes made prior to or during the course based on the teachers’ observations.

4.1 Clarity of marking

As students’ efforts are intermingled within group efforts that are further intermingled with the efforts of other groups, students in 2011 were particularly concerned how their work could be fairly marked. Would key contributions, or technical difficulties that might have required extensive thought but few lines of code to overcome, be overlooked? In the course feedback survey in 2011, clarity of assessment requirements was the category in which the course scored the poorest. 63% of students regarded the requirements as unclear, with only 11% considering them clear.

In 2012, we introduced a Demo Day in the final tutorial session, which as well as requiring groups to show us their code working also gave them a formal opportunity to pitch to the marking panel what their achievements were and how they could be found. Groups were also asked to provide a short pitch sheet describing their major achievements, the difficulties they had overcome, and where evidence for these achievements could be found (such as the code tests, ticket numbers, and which other groups they had collaborated with). Additionally, the marking panel had a checklist of questions to cover aspects that students might overlook as they describe their work to the panel. The groups’ pitches themselves were not marked, but were a guide to the marking panel when we reconvened to ensure that we did not overlook work that the students had done. As the results in section 6 show, the clarity of assessment requirements to students rose significantly, with 59% regarding them as clear (and 11% as unclear) though there is still room for improvement.

4.2 Using marking criteria to promote inter-group cooperation and earlier development

In 2011, we found that as there was no non-peer customer demanding progress, groups were not held to account to make progress until close to the deadline. The commit chart and newsgroup activity showed a significant spike in the final week of development. We also found that, except in the final week, groups interacted with each others’ code first and with each other second – there was little cooperation in the design of interfaces and functionality in the early weeks of the project.

In 2012, we made experimental changes to the marking criteria to see whether this would be enough to change students’ behaviour. The criteria were released in two stages – an initial broad set of criteria available at the start of semester (Figure 1), followed by a more detailed mark breakdown late in semester. The groups’ ticket management was extracted into a separate category of assessment, worth 5% of the course mark, with a requirement to keep the ticket updated as the groups progressed.

The panel will consider factors including:

- The quality and value of the group’s software improvements and contribution to the project
- The difficulty of any problems encountered during development, and how well the team managed to overcome those problems
• Whether the team's work, manner of work, and changes to the codebase assisted or hindered the work of other teams.

• Evidence of consistency of effort throughout semester

• Evidence that the team has been putting into practice techniques taught during the course.

Figure 1. Marking criteria description for the project work

The comparative commit charts are shown in Figure 2. The chart from 2011 has been transposed in time to make the due dates for the project coincide – the due date in 2012 was one week later than in 2011. The marking criteria alone were not enough to prevent groups from leaving their development late, although the peak is less pronounced than in 2011. The early peak at week 3 (adjusted) in the 2011 chart is from an individual programming assignment, writing a robot, which in 2012 was committed to a separate repository and is not reflected in the chart.

Looking at the individual commit profiles from 2012, the picture is not uniform across the class. Figure 3 shows the commit rates of the top eight most frequent committers according to GitHub. (The right hand cut-off is the project due date.) While students tend to share a late period of activity, many also have earlier peaks of activity. This suggests that while we can work to further reduce the tendency to leave development late, it is probably not possible to remove the late peak altogether – as the due date will always tend to cause students’ last peak in activity to coincide.

4.3 Groups invent their own tickets

In 2011, we crafted 32 tickets that groups could choose from, in order to ensure that although groups had quite different goals there would be overlaps in the underlying changes they would need to make to the core code. This was to ensure that groups would have to collaborate in their development work. In 2012, we allowed students to invent their own features to implement, in the hope that enough overlap would happen naturally to encourage collaboration. The project timetable required groups to create their initial tickets in the third week of class, even if they had not fully refined their idea of what they wanted to work on. In the Monday of the fourth week, groups gave a brief (unmarked) verbal description of what they intended to work on to the class in the tutorial. This gave groups the opportunity to begin collaborating early, before they became deeply invested in the plans they were making, in the knowledge that collaboration with other groups was a significant part of their project mark.

The results were deeply encouraging. All nineteen groups by the end of semester could indicate ways in which they had collaborated with at least one other group. 78 tickets were created in all, with 338 comments. The distribution of these is shown in Figure 4. The three most frequently commented on tickets were a ticket to fix some failing tests (28 comments) created by the tutor, followed by two tickets discussing common functionality needed by many teams (27 and 23 comments) created by students.
In 2011, we reported that some students considered it unfair that their classmates could break their code, and there was an impression that giving students features that depended on each other was setting them up to fail. In 2012, while groups still sometimes broke each other’s code, there was not the same sentiment of the task being unfair. We suspect that the improved collaboration between groups, their freedom to select their own task, and the marking structure giving them greater opportunity to bring any difficulties they faced to the markers’ attention may have alleviated this concern.

4.4 Removal of student-delivered content
Our original course design required each group to give a twenty-minute live technical demonstration applying a relevant tool or principle. These technical talks, designed to mimic a technical conference and employ a teach-to-learn pedagogy, delivered 20 of the 32 technical topics on the course. We reported that some groups had reported the parallel task of preparing a technical talk to be distracting from their project work, although this was not corroborated by the empirical commit data. When the official feedback surveys were released, we found these talks had been heavily criticised, with many of the feedback forms specifically criticising them and only 1 positive comment for them. Students lacked confidence in material delivered by other students, and as the survey was conducted at the end of a six-week block of student technical talks, a handful of students explicitly commented that the course teachers were not teaching.

In 2012, the student technical presentations were replaced by live technical demonstrations delivered by staff, with a focus on class interactivity and responding to questions. We used a revised version of [author’s system] to support live polls, anonymous chat, and to be able to paste console output or post-recorded video from the demonstrations into the online notes.

This gave us a greater opportunity to react to students’ needs in the course, but having to replace so much of the content at short notice (for administrative reasons, we had little lead time before the course began), many of the demonstrations were underprepared. We found that when we re-recorded two of the demonstrations to paste into the notes, we were able to make them shorter and clearer than the live performance. In future iterations, we recommend a mixture between pre-recorded demonstrations of tools that students can view out of class (inverted pedagogy) and live interactive explorations and demonstrations around issues the class is facing in their development.

4.5 Changes to examination and tutorial
We reported in 2011 that the textbook content was not well integrated into the course. This also caused the exam to be disconnected, as the textbook material had featured heavily. In 2012, we redesigned the exam around a number of problems, such as class design and debugging. We also introduced a mid-semester test to give students an earlier gauge of their progress.

As the semester progressed, we observed that some of the examinable material that did not relate directly to development tools, particularly design patterns and class diagrams, was not being exercised by students in their project work. As the tutors were billing us for less preparation time than for a regular tutorial, we had the available teaching capacity to introduce an optional tutorial sheet from week 8. This would also serve to acquaint students with the style of question they could expect on the exam.

We received anecdotal support for the tutorials – they appeared to be appreciated but there were requests for more structure in the tutorials earlier in the course, perhaps because late in the course students are more concerned with their project being due. A style of question that was particularly well received was a scenario question, presenting students with a development situation (for example, discovering the overnight tests have broken, or various kinds of bug). Students were asked how they would go about identifying and resolving the issue, using all the course tools at hand. We were approached by one student thanking us for this mid-semester test question as thinking through his answer had helped him to understand what he needed to be doing in the course.

5. LIMITATIONS DUE TO LEAD TIME
For administrative reasons, the course coordinator was only allocated to the course at a month’s notice. This meant that some of the content was rushed in its preparation, and that some changes we had suggested in our previous paper could not be implemented in this iteration.

In 2011, we found that although an imperfect legacy code base was necessary for the course, some aspects of Robocode were particularly hard for students to work with. Shortcomings in its unit testing made it difficult for students to understand what changes would or would not cause it to fail. Some core multi-thousand-line-of-code classes that students needed to modify were simply too large and entangled for students to want to take on the task of redesigning them in a better way – instead they mimicked the existing design and threaded yet more functionality through these classes. Some aspects of the code base had made mocking functionality difficult, making it hard for students to write appropriate unit tests for their own work.

While we had recommended switching to a friendlier codebase, or refactoring a subset of Robocode in advance of the course, the short lead time for the course did not allow this. This is a recurrent issue for the course, as Robocode had also been chosen in 2011 partly because we did not have the lead-time to consider many alternatives or to inspect the code base in any detail. While the course is not unduly demanding of staff time during semester, a larger amount of preparation is required setting up the systems and project before the course begins.

5.1 Change from Subversion to Git
In 2012 the version control system for the course was changed from Subversion to Git, with the code stored in a private repository on GitHub. Git’s lightweight branching would allow students to quickly push their incomplete changes to a private branch if students needed to move incomplete work from a lab computer to their own computer at the end of a session, or if they needed to enlist the help of a tutor remotely. By the end of the project, 31 branches were created by students, of which 18 were merged into master.

However, we found that Git was more complex to teach to the subset of students who had not encountered version control previously. In 2011, version control took less than an hour of lecture time to cover its basic use (a model of how it works, checking out code, making modifications, checking in code, resolving conflicts, and branches as directories in the source tree). In 2012 material on Git was introduced progressively across four lectures. The meaning of the hash code on commits, the directed acyclic graph of versions, remotes and remote-tracking branches, the difference between fast-forward and manual merges, the index (staging area), and the stash also needed to be covered. This placed a significant extra burden on the beginning of the course, as students get to grips with their development environment. It is
not clear whether the advantages of a more sophisticated version control system outweigh the extra time required to teach it.

6. COMPARATIVE STUDENT FEEDBACK
The comparative results from the official student feedback survey are given in Table 1. Table 2 was produced by classifying students’ written statements on the 2012 feedback forms.

Table 1. Student feedback summary, on a five point Likert scale (1 = strongly disagree; 5 = strongly agree)

<table>
<thead>
<tr>
<th></th>
<th>2011 27 responses, 72 enrolled</th>
<th>2012 27 responses, 66 enrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>I had a clear understanding of the aims and goals of the course</td>
<td>score 2.77, 31%</td>
<td>score 3.74, 63%</td>
</tr>
<tr>
<td>The course was intellectually stimulating</td>
<td>score 3.67, 70%</td>
<td>score 3.93, 74%</td>
</tr>
<tr>
<td>The course was well structured</td>
<td>score 2.59, 26%</td>
<td>score 3.26, 41%</td>
</tr>
<tr>
<td>The learning materials assisted me in this course</td>
<td>score 3.11, 37%</td>
<td>score 3.26, 41%</td>
</tr>
<tr>
<td>Assessment requirements were made clear to me</td>
<td>score 2.19, 11%</td>
<td>score 3.59, 59%</td>
</tr>
<tr>
<td>I received helpful feedback on how I was going in the course</td>
<td>score 3.11, 37%</td>
<td>score 3.07, 37%</td>
</tr>
<tr>
<td>I learned a lot in this course</td>
<td>score 3.59, 52%</td>
<td>score 3.96, 63%</td>
</tr>
<tr>
<td>Overall, how would you rate this course? (1=poor; 5=excellent)</td>
<td>score 3.00, 33%</td>
<td>score 3.59, 52%</td>
</tr>
</tbody>
</table>

Table 2. Classification of the most frequent comments on student feedback forms in 2012

<table>
<thead>
<tr>
<th>The best aspects of the course or teachers’ teaching styles (combined).</th>
<th>Suggested improvements for the course or teachers’ teaching styles (combined).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project mimicked real world</td>
<td>Better preparation of demos/lectures</td>
</tr>
<tr>
<td>Qualities or abilities of the lecturers</td>
<td>Issues with the lecture materials (printing issues, and detail)</td>
</tr>
<tr>
<td>Interactive teaching technology in class</td>
<td>More detailed marking criteria</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

The most concerning numbers from the 2011 survey related to the assessment requirements, student understanding of the course aims and goals, the course structuring, and the clarity of assessment requirements. All of these improved markedly in 2012 in terms of both average score and agreement. This is likely due in large part to the simplification of assessment materials, the removal of the parallel assessment items, and a focus on better communication of expectations. All other questions improved except the question regarding feedback, perhaps because of the removal of the student-delivered lectures, which incorporated a strong feedback component. The comments given by students highlight preparation of demos and lectures as a point of improvement. This suggests that improving delivery of the course might be a more significant focus going forward than further changes to the course structure.

7. TIMING OF STUDENT INTERACTIONS
The student commit activity, shown in Figures 5 and 6, is well distributed throughout the week and throughout the hours of the day. Only 18% (242 of 1332) of commits occurred during the scheduled tutorial hours (Mondays, 10am to 2pm), the remainder occurring in non-contact hours.

27 of the 66 students were present to respond to the official student feedback survey in Week 11. This attendance rate is typical of many on-campus classes. Video recordings of lectures are accessible online, and for this course the majority of technical demonstrations were also posted into the online notes in [author’s system] either as console output with accompanying explanations or as post-recorded YouTube videos. Accordingly, it could be argued that many on-campus students are effectively taking the course virtually rather than physically.

8. SUPPORTING VARIED COHORTS
In 2013, the cohort for course will change, as design students and database students will join the programming-oriented students. Not only does this mean that there are two additional categories of student to cater for, but it is also likely to create far greater differentiation between groups. There is no guarantee that there will be enough design or database students to distribute their skills uniformly across the groups.

There is an additional challenge, then, of ensuring that the project design can support a non-uniform distribution of skills. For
example, we do not believe that Robocode offers sufficient scope for interaction design or information management for students with an interest in those areas.

In 2011, the student groups came up with a convention that each of their features included an enabling flag, and that their functionality was turned off by default. In 2012, one of the groups took it upon themselves to create a “modes” API, that allowed other groups to choose whether their functionality would be a separate mode within the game or whether it could be mixed in to any other group’s mode. In this way, both cohorts developed conventions whereby they shared changes to core code, but could separate their functionality into substantially different products. (For instance, “robot soccer” needed to be a separate mode from “robot racing”.)

One way to handle heterogeneous groups, then, would be to formalise this distinction between common code and separate products. Rather than provide a single cohesive product, such as Robocode, that students edit into, we would provide two or three imperfect and incomplete libraries in a shared repository that groups could build out products from. In the early weeks of the project, groups would be encouraged to invent and cluster around a small number of product ideas, with the aim that many groups will cooperate building different features of a shared product, and that all groups will need to cooperate in terms of changes to the common core libraries. So long as the course enforces a common build system, the same continuous integration systems can manage and monitor all the projects with little extra effort.

This would have a number of advantages. Groups could take on a diversity of roles in a diversity of projects, depending on their makeup. As the common code base would be smaller, the class can move from understanding the existing system to ideation more quickly, and may be more willing to redesign and refactor the code rather than merely patch their work into it. It would also resolve the issue that the current practice of jimmying together the code rather than merely patch their work into it. It would also formalise this distinction between common code and separate products. Rather than provide a single cohesive product, such as Robocode, that students edit into, we would provide two or three imperfect and incomplete libraries in a shared repository that groups could build out products from. In the early weeks of the project, groups would be encouraged to invent and cluster around a small number of product ideas, with the aim that many groups will cooperate building different features of a shared product, and that all groups will need to cooperate in terms of changes to the common core libraries. So long as the course enforces a common build system, the same continuous integration systems can manage and monitor all the projects with little extra effort.

9. CONCLUSION

On the whole, the changes made to the course have been effective, with student evaluation responses showing marked improvements in areas that were problematic in the previous year. Students’ concerns in 2012 were predominantly around delivery issues – such as the preparedness of demonstrations, printing issues and detail in the notes, and providing structured material in tutorials earlier – rather than further problems in the course structure. Allowing the students to choose their own project topics was a success. There were no discernable disadvantages in terms of meeting learning objectives, and the self-determination likely mitigated students’ views about being “set up” to encounter problems, and made for a more organic appreciation of the tools and techniques being presented.

The more diversified cohort will pose challenges for the future editions of the course, in terms of group formation and finding projects that are sufficiently broad to challenge all students, and sufficiently overlapping as to necessitate intra- and inter-group collaboration. Attempting to open parts of the course up to online delivery will also present a challenge.

10. ACKNOWLEDGMENTS

We gratefully acknowledge the assistance of our course tutors, Jackson Gatenby and Tyson Henning. We also acknowledge 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.

11. REFERENCES

[1] Süß, J.G., and Billingsley, W. 2012. Using continuous integration of code and content to teach software engineering with limited resources. 34th International Conference on Software Engineering (ICSE), 1175-1184