Using Continuous Integration of Code and Content to Teach Software Engineering with Limited Resources

Jörn Guy Süß  
The University of Queensland  
School of ITEE  
Brisbane, Australia  
jgsuess@acm.org

William Billingsley  
NICTA (National ICT Australia Ltd)  
The University of Queensland  
Brisbane, Australia  
William.Billingsley@nicta.com.au

Abstract—Previous courses addressing the gap between student and professional programming practice have either isolated small groups’ development in such a way that larger scale difficulties that motivate many professional practices do not arise, or have required significant additional staffing that would be expensive to provide in a large cohort core undergraduate software engineering course. We describe the first iteration of a course that enabled 73 students to work together to improve a large common legacy code base using professional practices and tools, staffed only by two lecturers and two undergraduate students employed as part-time tutors. The course relies on continuous integration and automated metrics, that coalesce frequently updated information in a manner that is visible to students and can be monitored by a small number of staff. The course is supported by a just-in-time teaching programme of thirty-two technical topics. We describe the constraints that determined the design of the course, and quantitative and qualitative data from the first iteration of the course.

Keywords—Continuous Integration, Software Engineering, Studio Course, Resource Constraints, Experience Report

I. INTRODUCTION

It is a long-established problem that the experience students have of software development during their undergraduate studies is very different from industrial practice. Many universities have developed courses over the last ten to fifteen years that have attempted to replicate industrial practices in the classroom to bridge this gap between student and professional experience. In these courses, we have found there is a tension between the scale and realism of the programming experience and the amount of staffing and other difficult-to-provide resources that are needed to support that experience. Previous courses have either:

- separated each small group of students into a distinct codebases, where they do not face the issues of scale that motivate many of the practices – e.g. someone outside the immediate small group they meet with face-to-face breaking their code, or
- required so much extra staffing and additional rare resources that it would be difficult and expensive to scale to teaching the whole cohort of a core undergraduate engineering subject that way – e.g., needing to employ large numbers of senior developers of an open source project as teaching assistants.

In this paper we describe the first iteration of a course we designed to address this issue. All 73 students worked together to improve a large common legacy code base, staffed only by two lecturers and two undergraduate students employed as part-time tutors. Empirically, students did encounter difficulties of scale that we hoped they would.

The course is designed around continuous integration and automated metrics, openly available to staff and students alike. A rich set of regularly updated data is collected and coalesced into a small number of channels so that it can be monitored by a small number of staff. This includes taking measures to ensure that the staff can observe the student conversation, including encouraging them to engage in anonymous chat in lectures. A just-in-time teaching programme covers thirty-two technical topics just when students are feeling the pain of needing those tools and techniques.

We describe how the course design is derived from the constraints that it must satisfy. We present quantitative and qualitative empirical results from the first iteration of the course in practice, revealing many of the issues that arise in teaching a class in this way. We believe that our course shows that valuable experience of larger-scale software practices can be taught without incurring significant extra staffing costs within a well-fitting single semester teaching subject.

II. CATEGORISING THE PROBLEM

Unless a student participates in an open-source project or works in a company, it is likely their experience of software development during studies will be limited to a few small academic examples. And while software engineering courses teach many theories and fundamental computer science concepts, such study is largely scholastic in nature. This leaves students unprepared for their professional roles as the gap between the scholastic examples and the professional need to write real-world software reliably and at scale is large [1]. As
educators, we identify seven differences between education and practice that we are particularly interested in.

- **Toy Examples.** Students generally write “toy” programs in short time spans, with few collaborators, and without legacy. These provide little experience of the issues that arise when developing software at a professional scale.

- **Assessed once, revised never.** Even larger examples are only superficially tested, graded and then discarded [1]. Unlike professional software, student assignments are not revised or refactored.

- **Intolerance of Ambiguity.** Typical student programming assignments are relatively unambiguously specified in detail with no requirements changes. There is little sense of students having to expend effort to discover what they need to do.

- **Abstraction versus Reflection.** It is important for university courses to focus on core competencies as abstract capabilities as opposed to specific skills like for example, any specific choice of software development methods [2]. But understanding of professional methodologies is difficult to teach theoretically and should be based on experience and reflection on one’s creative process [3]. This poses an obvious dilemma in terms of teaching goals.

- **Untaught Topics.** Engineers are naturally expected to be knowledgeable about numerous topics that do not typically appear on undergraduate syllabi. While these topics are often small practices rather than core computer science fundamentals, the knowledge gap can make students appear estranged from practice as they are unable to relate to the language and culture of software engineering.

- **Cultural differences between education and practice.** To be able to mark students individually, there is an ingrained culture of isolation in students’ work during their studies that makes an entire class collectively improving a software product culturally alien.

- **Dissimilar practices.** The practice that students observe in education does not match professional practice. Staff-provided software examples usually do not follow industry norms and standards. The university teaching environment often lacks fundamental software development tools that are minimum tooling even in community/open source projects. The tooling and manner in which assignments are set to students and submitted by them is alien to how software professionals will receive and deliver tasks.

Naively, these differences could be addressed if students were exposed to real software in a real setting. However, the constraints of a core undergraduate university teaching course raise a number of challenges:

- **Time:** A university course does not have the continuity and density of a commercial project. Even with a well-established development team on full-time work, the twelve weeks of a semester are a very short time to create a product.

- **Trunk:** Creating a new software from scratch does not reflect the industrial reality that 80% of the work is maintenance and enhancement. This leaves the challenge of how to obtain an existing software that offers both quality for reference and opportunity for improvement.

- **Reflection:** While the experience of a work process is worthwhile, the real goal must be for the students to obtain an insight that allows them to select and alter its elements in their future engineering practice. This requires additional study time that cuts into the fixed and limited work time budget.

- **Fairness:** If work is done in groups, then the selection of group members is a crucial success factor. How can we guarantee a level of fairness in this process?

- **Bootstrap:** Due to the semester time constraint groups will need to be productive from the start, although they are only learning about tools and practices during the run of the course. This creates the challenge of finding an order of learning that will allow such productivity.

### III. Related Work

Several courses have applied Extreme Programming or its practices [4] in undergraduate teaching. Many of these courses [5]–[8] have asked pairs or groups of students to undertake short separate assignment-like projects using pair programming, test-driven development, or automated testing practices. Other courses [9], [10] have made continuous integration infrastructure available for longer group projects, but again with groups working on separate greenfield projects. In these smaller-scale cases, as groups work independently of one another, many of the issues involved in professional development do not arise. Refactoring tends not to occur, and every user and every author of every line of code in the entire history of the project regularly meet face-to-face. This means that issues of intent, design, documentation, communication, and coordination can be mediated verbally rather than needing to understand the technical practices that are needed in larger and longer developments. Some courses have asked students to engage with much larger projects, but these courses typically rely on relatively small cohorts of students, special resources, and significant extra staffing, making them difficult and expensive to run for all students in a core software engineering course at a major university.

Allen et al. [1] designed a software engineering course at Rice University in which students worked on DrJava, an in-house open source project, using Extreme Programming methods. The three most senior members of the DrJava development team were employed as teaching assistants, each managing a team of two to six students to implement
a development task. Of the three features only one was completed and it had a significant bug that was not revealed by the students’ testing. Nonetheless the course was deemed a success, and the authors concluded that “our production programming course can be replicated at other colleges and universities where funds are available to start a small open source programming project”. However, this model does not scale. Assuming the maximum group size of six students per manager, our cohort of 73 students would have required more than 12 of the most senior members of the open source project’s development team to be employed as teaching assistants; That is assuming the university has a funded open source project that is appropriate for inexperienced undergraduates to engage with.

Carnegie Mellon University has pioneered a successful “software studio” course for many years [11], [12]. The studio works like a small development company, with students working in groups of approximately 3 to 7 developers, plus a technical lead and a project lead, on real development problems for paying customers. The projects are typically greenfield developments, though as there are multiple student intakes, approximately half the students join an ongoing project. Junior students move into senior, technical lead, and project lead positions as new junior students join. The course takes up 40% of a Masters degree, running for sixteen months, including full-time work over a summer semester.

When exporting the software studio concept to other institutions, Root et al. [13] found a number of critical factors to the concept’s success. The program is “critically dependent on the quality of the students it accepts”, and “standards cannot be relaxed or problems will ensue”. Pennsylvania State University successfully implemented a shorter 14-week version of the course, but this was at postgraduate level and virtually all the students were full-time software professionals [14]. For a software studio of this type, external sponsors must be found that are not only willing to be closely involved in the projects, but that are willing to pay money for the development, as without charging money for the development it is too easy for a sponsor to pull out. Together with the unique and lengthy structure of the course, these factors make the software studio concept difficult to apply to undergraduate education, where cohorts are much larger, students are much less experienced, the course may be a mandatory subject that must be amenable to mediocre students as well, and time and resources are short. The projects themselves, greenfield projects of around seven developers, are again relatively small in terms of the number of developers. Consequently, some issues with larger projects and legacy code bases would not surface.

IV. COURSE HISTORY

We redesigned a course that had been run for a number of years. It was based on asking small groups of students to select, analyse, and extend an open source Java tool [15]. This course had been strongly influenced by the Carnegie Mellon Software Development Studio program, but many aspects of the course had been removed or simplified in order to apply it to a core undergraduate subject. The course could not provide external paying clients, could not afford to provide mentors for each group, ran over a single semester, and had a large undergraduate cohort rather than exclusively selecting a smaller cohort of high calibre postgraduates. Consequently, it suffered from many of the issues listed in CMU’s experience report [13] on exporting their studio approach to a partner university. Without the continual feedback of dedicated mentors, assessment cycles were long.

As the projects were selected by the students, no sufficient vetting process was available to ensure that the projects were suitable, and many groups ran into difficulties. Lecture material was disconnected from student practice and problems. The tutorials were used to present tools and practices, but due to the lack of mentors, these were demonstrated on small standard examples and their applicability was not demonstrated in the context of the actual problem. Finally, it was impossible to assess the state of the students software, as the single lecturer could never become familiar with the various different open source projects used. To validate the outcomes, students were required to produce report documents. While this form of assessment trained students in planning and writing skills, it drew the focus away from the software, which beyond a five minute demonstration was not formally assessed or reviewed. As noted earlier in the paper, this style of course also does not expose students to issues that arise in larger programming endeavours.

V. COURSE DESIGN

Our redesign of the course aimed to provide a quality teaching experience under human resource constraints. That is, we aim to address many of the seven issues we identified earlier in this paper without requiring large amounts of extra staffing. The course should give experience of the issues that arise in larger projects, involve refactoring and revision of code, require students to embark on a process of discovery, give opportunities for reflection on practice, cover a number of technical topics that are important to professional practice, encourage large-scale collaboration, and employ practices and tools that are similar to professional software development.

A. STUDENT BACKGROUND

The course is a single semester course, which ran for a cohort 73 students with two lecturers and two undergraduate students employed as part-time tutors. The prerequisite subjects for the course are two introductory courses teaching programming in Java.

The cohort of students is a mixture. Some are known to have been working as programmers prior to beginning their studies, while others’ experience consists solely of the
Students test coverage and quality of their code at each build. Analytic tools that would show students metrics about the students were developing in a manner that was visible and continually updated. Particularly important would be incorporating appropriate tools into the automated build the teaching content from lectures to student practice. By work. It also gave us a straightforward mechanism to apply the project, and also provided a visible history of students’ work. It turns out that while writing robots is easy, using the well-defined API of the game, altering the game is not. The code base is a genuine legacy system with a ten year development history, predating the tooling that is now used to build it. While we cannot greatly increase the student time-on-project in a single semester course, applying students to such a legacy project allows students to experience many of the issues that arise in long-running project.

This also means that even though students were given relatively clear features to implement, they need to embark on a process of discovery of the design of the existing code to work out how to implement them.

D. Inter-Group Collaboration

We allocated the students to work in 20 groups of three to four students. Each group chose a feature to implement,
from a selection of 30 features. However, it should be noted that these 20 groups were not working independently – Continuous Integration made all 73 students colleagues modifying a common code base even if they were not in the same group. Hence they could not simply rely on group knowledge of their plans and changes. Students’ code could be broken by changes made by other groups. The only way to defend code was to test it, so that others would become aware of the impact of their actions, and to communicate your intentions and actions in public forums and collaboration tools provided. We made it clear in the introductory lecture to the course that groups would need to communicate. The course would increase the maximum project size experienced by students to 73 for all students.

Many of the features offered were deliberately designed to promote cooperation between groups. For instance, one feature involved implementing elastic collisions for robots, while another feature involved adding walls as obstacles into the battlefield – suggesting that these features might need to interact to allow robots to elastically collide with the walls.

E. Continual Integration of Content

The course content includes thirty technical topics around professional methods and tools of software production. As the course must cater for very inexperienced students, the need to enable students to work productively on the project dictated the teaching order of many of the early topics. Essentially, the course had to implement a “Just-In-Time” teaching programme – the topics students would learn in lectures related directly to the issues they were facing working with the code at that stage of the course. The subject matter had to be immediately put into practice by necessity.

For example, before students could begin to work with the code they needed to be able to check out the source code, build it, modify it, and know whether or not their modifications have broken it. Accordingly, the first topics taught needed to be version control, automated build systems, and unit testing. For Robocode this meant that they had to learn the basics of Subversion, Maven, and JUnit by the third week of semester. After development had started topics progressed to enable the students to

- discuss design in terms of class and sequence diagrams,
- discover the causes of unexpected behaviour using debugging, logging, and profiling,
- understand and create good design using metrics, design patterns, and refactoring, and
- test designs effectively using testing patterns and object mocking.

Nearer the end of the course topics were less constrained by necessity. They were intended to present an outlook of current challenging topics like polyglot programming, plug-in infrastructures, cloud computing, and platform-as-a-service. We included these topics to foster awareness of these topics, as they all featured in our course. For example, Robocode robots can be programmed in several languages, Robocode is based on an inversion-of-control container, and our Continuous Integration system was run on an Amazon cloud service.

Overall the course covered 32 technical talks over 12 weeks. This is a much greater breadth of topics than in would appear in a typical lecture course.

F. Student-Delivered Content

We were aware that mere presentation or lecturing of content is one of the least effective methods of teaching [18], and sought a method of ensuring that the lecture slot for the course furthered our goal of getting students to experience professional practice. We decided to adopt a teach-to-learn pedagogy. New techniques and tools are often introduced to a company by engineers teaching their colleagues and even their managers about them. Engineers will also frequently have to explain and demonstrate their work to others. We asked students to present 20 of the 32 technical topics on the course, requiring them to both explain the topic and give a live technical demonstration applying a relevant tool or principle. The talks were limited to 20 minutes in length, and were essentially a shorter form of the kinds of talks that occur at professional conferences such as JavaOne and Devoxx.

Students presenting so much of the course content created a risk that they might present material poorly that was then examined at the end of semester. To mitigate this, students gave a dress rehearsal of the presentation to one of the lecturers some days prior to the lecture in order to gain corrective feedback, and we left space in the lecture timetable to re-present topics ourselves should this become necessary. We also used The Intelligent Book [19] as the presentation and content delivery mechanism. The Intelligent Book is a cloud teaching, learning and presentation platform that supports collaboration, interaction, and semantic annotation of content. Each slide of a student’s presentation was semantically tagged with a topic. As a plurality of content can exist against a topic, lecturers, tutors, or even other students could add additional resources, external content, exam practice questions, and other materials against the topics of students’ slides. The Intelligent Book also provides a chat stream that can be opened on students computers or the lecture screen alike. We employed this facility to provide anonymous chat on the lecture screen during the lecture, to be able to observe the questions and comments that students might not be game to ask in person, and to expose what was not being understood. The anonymous chat was frequently used by students to raise concerns or ask questions. However, we found even seemingly trivial comments such as *brain explodes* can be useful as they give an insight into how material is (or isn’t) being received. In one case, the chat bar uncovered plagiarism by posting
the accusation live during the lecture, including the source URL from where the material had been lifted.

G. Continuous Integration Infrastructure

Robocode’s source code repository was cloned (including all revisions) and made available to students as a course Subversion repository. Two continuous integration servers ran regular automated builds using Maven. One of these was a commercial cloud-hosted environment, JIRA Studio, kindly provided by Atlassian and integrated with their ticket management and agile planning software. The other was an instance of the Jenkins open source continuous integration server running on a virtual server within the university. (It would be possible to run the course with just one of these, but as the infrastructure was set up shortly before the start of semester, we created two environments in case one of them proved insufficient in some way.)

Sonar [20], an open source code quality management platform, was also installed on the virtual server and integrated with the Jenkins automated builds. Sonar augments the Maven build process with plug-ins that gather data such as test code coverage, static analytics measures, and coding standard violations, and collects the data to present to students in a Web-based dashboard interface.

The Maven build process of Robocode was augmented at various stages during the semester to provide tools that were discussed during lectures. These include automatic generation of class diagrams as part of the overnight builds, and adding an object mocking library that would be useful for unit testing.

This tool support, based largely on freely available open source software, fills three crucial roles:

- It provides continuous detailed feedback to students on the status of their and others’ work. They can see tests being added, code being refactored and moved, problem areas of the code base are highlighted, and it is relatively easy for students to discover who broke the build. Without this facility, 73 students would find it very hard to collaborate on a single code base.

- Through its necessity and usefulness it encourages students to interact with the very tools and techniques that are the technical topics on the course.

- It provides continually updated consolidated information on the whole class’s development to staff, with the ability to drill down easily into the work and changes of any group or student. This allows a small number of teaching staff to monitor the progress of a large class effectively.

H. Assessment Design

The previous paragraphs have alluded to the various tasks that students have to handle during the semester: Individually create and improve a battle robot, give a short talk on a software engineering subject, and create a new feature of a game server.

Students are motivated by assessment, and where possible we would recommend making the assessment visible and continuously updated, in the manner of the software metrics, so that students iterate on their own progress. Principles of continuous assessment have been advocated and explored by Penalvo et al. [21] amongst others. However, the task of assessing a student programmer on a realistic project is equivalent to the problem of a programmer’s annual review, and is always likely to involve human judgment. This resulted in the first iteration of the course in a somewhat complex markings structure, as shown in Figure 3.

Group work has a number of issues including social loafing and group composition. To address these issues, we followed established work by Kavanagh and Crosthwaite [22]

- our groups were pre-allocated to minimize the major risk factors of students speaking English as a second language, GPA imbalance and uneven competency in the programming languages.

- To dissuade social loafers from exploiting other group members, we applied a peer assessment scheme were the student marks for the group work are divided by the contribution that the students have agreed on.

- To avoid escalation late in the project we required groups to lay down ground rules for their group work that would be used by course staff for mediation in the case were group work issues arose. For all issues pertaining to group work and work methods we provided a dedicated tutor that the students could address themselves to without information being passed to the lecturers.

Presentations were marked through peer assessment, both as an incentive to pay attention and to improve students ability to judge material. This naturally leads to the problem of mark inflation: Why not assign the top score to all groups? We resolved this by double marking: The student marks are scaled by a lecturer’s mark. Assigning the top score to everyone does not improve the score, it will only level the differences. To get information on the lecturer’s prospective score for a group, the students could attend the dress rehearsal. We also required students to provide exam
questions drawn from their presentations. These questions were posted as training material and a subset was eventually included in the exam.

While the project work forces students to encounter many of the problems that result from poor programming practice and poor methodology, it cannot actually force students to learn.

VI. EMPIRICAL DATA

Figure 4 shows the graph of the number of commits to the source code repository across the twelve weeks of the course.

There was ambient activity of thirty to fifty commits per week for most of semester, followed by a final week rush that also included a significant amount of refactoring. In class we characterised this to students by suggesting that until week 11, students were not practicing continuous integration (groups were not frequently committing small chunks of work), and in week 11 they were. A similar pattern of activity is shown in Figure 5, which captures the number of posts on the course newsgroup per day.

This pattern of activity is less concerning than at first it might appear. Even in the fallow weeks, there was an average of one to three commits per group per week, and as the qualitative data will show, students were encountering issues around breaking each others’ code, finding poorly designed code difficult to work with and test. (To put the commit rate in perspective, when the source code was uploaded at the start of semester, its revision history contained 3756 revisions representing development from March 2005 to September 2011, at an average of 11 revisions per week. In twelve weeks, students committed a further 1536 revisions.) Week 11, meanwhile, gave all groups an intense experience of working on a large and rapidly evolving code base, as many core classes were refactored to make unit testing easier, including a number of instances of groups refactoring other groups’ work.

Figure 6 shows the total number of commits across the semester for each group. The activity distribution was also very distinctive. In groups that had low numbers of commits the commits tended to be concentrated with one committer, while in groups with high numbers several committers were active. As far as we can tell at this time, there is a correlation between the outcomes of the project and the commit habits of the students.

On average, groups committed 76 revisions, which is 75 more increments than they would submit in most assignments.

In the final lecture of the course, a number of straw polls were taken to gain a sense of what students experienced during the class. The results are shown in Table I. Again, these are responded by self-selected students who brought an Internet-capable device to class, so there may be selection factors in the results. The last three results indicate that while not every student experiences every difficulty, through reflecting on their collective endeavour as a single large cooperating cohort, they do gain valuable experience of programming at a larger scale.

VII. QUALITATIVE OBSERVATIONS

Student feedback was collected via newsgroups, questions, and chat comments throughout the semester. We also received feedback from students through a standard lecture survey that was evaluated at the end of the semester. The
Table 1
STRAW POLL RESULTS OF THE FINAL LECTURE.

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
<th>&quot;No answer&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...someone hindered me by breaking the build</td>
<td>14</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...someone hindered me by breaking my code</td>
<td>5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>...I broke the build</td>
<td>6</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>...I was hindered by a badly designed class</td>
<td>16</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...I was hindered by a lack of unit tests</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>...I was hindered by a lack of documentation</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>...I learned a greater appreciation of the importance of good unit testing</td>
<td>17</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...I learned a greater appreciation of the importance of good class design</td>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>...I learned a greater appreciation of the importance of continuous integration</td>
<td>16</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

following observations are based on the collected student feedback.

A. Tension between letting students experience problems and marking equity

Some students expressed concern that it was unfair that classmates could break their code, and that giving groups features that might depend on each other was setting them up to fail. These comments were responded to by other students commenting that it gave them an idea of what it was like to work on a real system and with badly written code and that it got them to practice working together. As lecturers we were happy to see these discussions, and the frequent posts on the newsgroup telling various groups they had broken the build, as this indicated to us that the students were experiencing the issues of larger scale development, such as the fact that their code and their actions impact on students outside their own small group. The first occurrences of students posting that other students had broken the build came as early as week 4. Our plan was to allow students to experience and hopefully overcome these issues, while then ensuring equity in the final marking. However, this distinction should be made clearer to students.

B. Clarity in the marking scheme is difficult to achieve early

As students efforts would be intermingled within group efforts that would be intermingled within the efforts of the class as a whole, the marking was always likely to be complex – we liken it to the more general problem of rating a professional programmer in their annual performance review. As the course had been significantly revised we did not want to commit ourselves to a specific detailed marking scheme until we had seen at least some progress made by students to give us confidence our chosen scheme would work. However, many students are motivated by assessment, and the late release of the marking criteria added to their uncertainty over whether the assessment of their personal work could be fair when their work is so deeply entangled with the work of their classmates.

C. Students inhibited by issues in the code base and mimicking bad design

The legacy code base was expected to provide avenues for students to refactor code, increase test coverage, and make other improvements. In this sense, an imperfect legacy code base was considered to be a virtue. In practice, some aspects of Robocode created greater than expected difficulties, and while these were generally overcome and gave students an increased appreciation of the importance of good design and testing, they made the student experience more painful than it needed to be.

In the early weeks of the project, students faced a lack of good unit testing in the codebase, as Robocode is tested with a number of integration tests that run full game simulations with predefined tanks. The outcomes of these game plays are assessed for test results. Since Robocode is multi-threaded, tests would fail non-deterministically in many cases. This made it difficult for students to understand what changes would or would not cause the software to fail. An issue in the way the unit tests are run within Robocode also impacted on the build system – Sonar was unable to collate test coverage from the integration tests, and also misreported builds as failures in Jenkins. This made working with the course infrastructure more complex for students, as they had to understand how to work around these issues.

Students also ran up against the issue of two large final classes, RobotPeer and Battle, which referenced each other and had poorly documented interfaces, forming a multi-thousand line of code monolith that was difficult to test but that students needed to modify. As late as week 9, groups were found to have been mimicking the poor structure of these classes, extending them with yet more functionality for which no interface was defined and yet more hard-coded dependencies between concrete classes. In the final two weeks of the project, students took a significant effort to refactor these classes, including refactoring each others’ additions where necessary, but a slightly friendlier code base might have made it easier for students to engage in this valuable refactoring sooner.

In order to prevent robot writers overstepping their boundaries and hacking the game, Robocode includes a safety
mechanism that is active when robots are loaded. This mechanism made the system difficult to test for students, as it prevented the straightforward use of object mocking libraries.

D. Groups interacted with each others’ code, but did not interact enough with each other

Students reported that he groups were attempting to implement their functionality separately, and were not being called into account enough for their actions in the code base. Generally, we noticed that groups interacted with each others’ code first, and with each other second. There were many newsgroup interactions about how one group’s code had broken the build or affected another group. There were some newsgroup interactions whereby a student would inform another group that he or she had modified their code. However, there were few cases of groups informing each other that they were going to refactor their code, or discussing how, or co-operating in the design of interfaces and functionality. In the final week of development, as activity spiked, student cooperation improved markedly. For instance, the course Sonar install failed close to the deadline, and students took it upon themselves to make code test coverage reports available to the class as a whole.

E. Chronologically long development cycle

Some comments were received that ten weeks is far too long a single development cycle. This is a difficult problem to mitigate. Students are only part-time on the project (they have other courses), and their development work is only two to three weeks full-time equivalent, which is a reasonable cycle length. As student’s loads in other courses are unpredictable, imposing intermediate deadlines might unduly punish students with “lumpy” workloads in their other studies.

F. Request for non-peer subjective feedback

Although students could observe each other’s progress and working, there was concern that as there is no non-peer customer demanding progress, they are not held to account to make progress until close to the deadline. A student suggested that perhaps they should have to show their progress to an external client every couple of weeks, though this would require significant additional resources. We suspect that this issue might be possible to mitigate without additional resources simply by having the existing lecturers pay more diligent attention to the activity charts during semester – calling to account groups that do not appear to be showing sufficient activity more regularly.

G. Parallel assessment items were reported to be distracting

Some groups reported that they were focused on the presentation they needed to deliver until after their lecture slot, and then switched to focusing on the development. This is not entirely corroborated by the empirical data, as superficially commit activity does not appear to be dependent on when during semester groups delivered their presentation – there was not a noticeable uptick for each group after their scheduled slot. We suspect that as all courses are in competition for students’ time, eliminating the presentation would not necessarily cause students to devote extra time to their development work.

H. Textbook content was not well-integrated into the course

The students were expected to read sections of a book on Java programming practices each week, which are examinable material. However, unlike the lecture topics, these were not well-integrated into students’ work.

VIII. Future Work

At the time of this writing our course is entering its final weeks, and we are looking forward to correlating students’ patterns of working with their final results. Based on the information already gathered, there are a number of avenues of improvement we can recommend for future iterations of this kind of course:

- investigate methods to improve the ways in which students are called to account during the course, to encourage greater participation earlier in the course
- make improvements to the build system and code base before the semester begins, so that the starting difficulty for the group work is reduced.
- extend the environment to provide continuous deployment, to provide a truer feeling of product development.
- define a subset of the code base for students to use to provide better control over the integration of the features and avoid imitation of bad design practice contained in the code.

Finally, further work would still be helpful in developing ways to encourage students who are effectively low-FTE part-time workers with varying schedules to cooperate and communicate as they design their code.

IX. Conclusion

Previous courses addressing the gap between student and professional programming practice have either isolated small groups’ development in such a way that larger scale difficulties that motivate many professional practices do not arise, or have required significant additional staffing that would be expensive to provide in a large cohort core undergraduate software engineering course. The course described in this paper enabled 73 students to work together to improve a common large legacy code base, collectively encountering and handling many difficulties that arise in large development efforts. The class was staffed by just two lecturers and two undergraduates employed as part-time students.

The success of the course is owed to continuous integration practices and automated metrics that coalesced
frequently updated information about the status of the class’s progress in a manner that was visible to students and could be monitored by a small number of teaching staff. Practices such as encouraging anonymous on-screen chat during lectures further supported the staff in monitoring the class’s progress and difficulties, by enabling staff to observe the student conversation. A just-in-time teaching programme supported the development by covering thirty-two technical topics timed to coincide with when students needed those tools and techniques to resolve issues they were facing in their development.

Empirical data suggests that the class encountered many issues that arise in larger development efforts, and while not every student struck every issue, through experiencing and reflecting on the collective effort, students gained valuable understanding of the issues and how the professional practices and tools they had been taught are used to address those issues.

There remain a number of avenues for improving the course design, but we believe this course has demonstrated that it is possible to provide valuable experience of large-scale development issues and professional practices within the typical resource and time constraints of a large cohort core undergraduate software engineering course.

**REFERENCES**


