View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Bournemouth University Research Online

Redesigning an Undergraduate Software Engineering Course for a Large Cohort

Claudia Iacob
School of Computing
University of Portsmouth
Portsmouth, United Kingdom
iacob@port.ac.uk

Shamal Faily
Department of Computing and Informatics
Bournemouth University
Poole, United Kingdom
sfaily@bournemouth.ac.uk

ABSTRACT

Teaching Software Engineering on an undergraduate programme is challenging, particularly when dealing with large numbers of students. On one hand, a strong understanding of software and good programming skills are prerequisites. On the other hand, the scale of the projects developed as part of undergraduate programmes do not always make the need for engineering obvious. Encouraging teamwork when students have little professional experience also adds to the level of complexity when delivering material. In this paper, we present a study on the redesign of a second year undergraduate course on Software Engineering for a large cohort.

CCS CONCEPTS

• **Applied computing** → *Collaborative learning*; • **Software and its engineering** → *Contextual software domains*; *Software design engineering*;

KEYWORDS

Software engineering, teamwork, education

ACM Reference Format:
Claudia Iacob and Shamal Faily. 2018. Redesigning an Undergraduate Software Engineering Course for a Large Cohort. In *ICSE-SEET'18: 40th International Conference on Software Engineering: Software Engineering Education and Training Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3183377.3183381

1 INTRODUCTION

Teaching Software Engineering (SE) as part of a second year undergraduate course at university is a challenging task for several reasons; these range from the nature of the topic, to the background and experience of the students. SE, as a discipline, emphasises the processes and methods put in place to develop complex software. It requires a deep understanding of the nature of software, and good programming skills. The need for structured processes and methods becomes obvious only after one has experienced the development process of a complex system. Reaching the second year, most students have only been exposed to developing toy software systems, or solving independent exercises. Consequently, the rationale for imposing a cumbersome and lengthy process for solving what appears to be a simple problem may not be entirely obvious.

For most second year students, software artefacts are analogous with source code, leading to the misconception that software development in general is equivalent to coding. The challenge of making students look beyond this, and encouraging them to document every step of the process is non-trivial. It requires building the context that will expose students to the real-world consequences of hacking rather than engineering software. However, the level of complexity of any second year assignment can rarely reach the state where such consequences are immediately obvious.

A crucial component of software development today is team work, and this needs to be reflected in the design of Software Engineering courses. However, working as part of a team is, for many second year students, a completely new experience, and comes with its share of challenges, e.g. dividing and synchronising work among members of the team, intra-team communication, scheduling team work around deadlines, and managing risks. Finding the balance between the right design of the team work element and managing the challenges teamwork brings requires significant planning. Moreover, growing cohort sizes in Software Engineering units means that any course innovation is risky, particularly given the lack of study examples from which best practice or lessons learned can be drawn.

For the past three years, we have delivered a Software Engineering course for second year undergraduates. This is a mandatory course for almost all undergraduates in the year, and – to ensure the degree programme meets the standards set for national accreditation – needs to include a team-based coursework project. As Table 1 illustrates, enrolment for this course has increased annually, and the growing number of students has brought additional challenges:

C1: Ensuring consistency in assessing individual coursework reports and marks,

C2: Managing teamwork across an increasing number of teams,

C3: Ensuring interactivity and student engagement,

C4: Providing students with comprehensive feedback on their work,

C5: Supporting students in developing the skills required for completing various phases of software development.

To meet these challenges, we present a study on the redesign of an undergraduate Software Engineering course for a large cohort. The redesign took place during the 2016/2017 academic year, and we redesigned the course around the following six principles:

P1: Freedom of choice. Teams had the total freedom over the development of their coursework project (C3).

Permission to make digital or hard copies of part or all 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. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE-SEET'18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5660-2/18/05.
https://doi.org/10.1145/3183377.3183381

Table 1: Student enrolment, Student feedback response, Coursework mean mark (standard deviation), Exam mean mark (standard deviation)

Year	Enrolment	Responses	Coursework	Exam
2014/2015	142	30 (21.1%)	39 (20.2)	46.2 (13.2)
2015/2016	177	93 (52.5%)	40.8 (19.2)	55.5 (10.8)
2016/2017	191	44 (23%)	54.2 (17.5)	43.4 (14.5)

P2: Focused assignments. Teams were required to submit specific deliverables focused on each stage of the coursework project (C1, C3).

P3: Peer assessment. Teams were fully involved in assessing each others work (C1, C2, C4).

P4: Relatable resources. Demos of software development tools were made available (C5).

P5: Software Engineering immersion. Practical sessions were designed to simulate real-world phases of software development (C5).

P6: Mediation not confrontation. All conflicts that arose within the teams were resolved by the course coordinator through mediation (C2).

We provide an overview of the Software Engineering course in Section 2. We detail the changes applied to the course during the redesign in Section 3, before presenting the results obtained in Section 4. We discuss the changes as they were perceived by students in Sections 5, and describe the related work that motivated our approach in Section 6. We conclude with the main take-home points in Section 8.

2 COURSE STRUCTURE AND CONTENT

The redesigned course is an undergraduate introductory course in Software Engineering. The aim of the course is to familiarise undergraduate students with the technical and process problems encountered in developing large scale and diverse software systems. As prerequisite for the course enrolment, all students would have had to pass an undergraduate level course on programming. The course was scheduled over two teaching blocks (TT1 and TT2). Each teaching block contains 12 weeks, with one reading week designed for independent study scheduled during each teaching block.

2.1 Before Redesign

During the first two years, the course was organised around weekly one-hour lectures, and fortnightly one-hour practical sessions. The assessment consisted of a written exam, and a group coursework project; each contributed 50% of the final mark. The project asked students to work in teams to develop a specific software application. All teams worked on the same application. A brief description of this application was provided and examples of this included a Gantt chart management application (2015/2016), and an appointment scheduler (2014/2015).

The coursework was designed around three submissions: 1) a team presentation of the project idea and a team project plan report (mid TT1), weighting 10% of the coursework mark), 2) an individual reflective report describing personal reflections on the project and the process followed, and the student's contribution to the project (end TT2), weighting 70% of the coursework mark, and 3) a team live demonstration of the system (end TT2), weighting 20% of the coursework mark.

The lectures covered a single topic each week. These topics included Lifecycle models, Requirements elicitation and specification, Prototyping, Estimates, Design (TB1), and Object-oriented SE, Documentation standards, Configuration management, Development environments, Project management, Software quality, SE for Embedded, real-time, and web systems (TB2). All practical sessions were designed to discuss coursework progress, and were organised as talk shops around each team's work on the coursework, common concerns across teams, and tips on techniques tried for various phases of the development process. Given that all teams worked on developing the same software system, all students could relate to the issues brought up by each team.

2.2 After Redesign

The third year of running this course saw a significant overhaul due to the increasing number of students enrolled in the course. The course remained a 24 week course, split into two teaching blocks with one reading week scheduled during each teaching block. The reading week was designed for students to catch up with reading and work on the coursework. Office hours were scheduled during these weeks, and students were encouraged to use these hours to clarify any issues with the coursework or the teamwork.

The assessment criteria remained unchanged, with the final mark being calculated as the average of the exam mark and a group project coursework mark. The coursework asked students to work in teams and develop a medium size software system. As opposed to the previous two years, each team got to work on a different system with no two teams developing the same application. Other coursework related changes incorporated in the redesigned course included: defining precise deliverables and corresponding marking schemes, designing all deliverables as group submissions, and allowing students to choose the system they want to develop. These changes are detailed in Section 3.

A one hour lecture and a one hour tutorial were scheduled every week. The lectures were split into 4 clusters, as follows:

1) Cluster 1 (the first half of TB1) covered Software Development Lifecycle Models (week 1.1) and Requirements engineering (weeks 1.4 and 1.5),

2) Cluster 2 (the second half of TB1) covered software design (weeks 1.8-1.10),

3) Cluster 3 (the first half of TB2) covered software implementation (weeks 2.1-2.4), and

Table 2: Redesigned course structure and content: weekly lecture and practical topics

Week	Lecture	Practical
1.1	Introduction to Software Engineering	Coursework introduction
1.2	Software Development Lifecycle Models	Project planning and management talk shop
1.3	London Ambulance Service (case study) [10]	Coursework review
1.4	Requirements engineering I (Elicitation)	Peer review assessment session
1.5	Requirements engineering II (Specification and Validation)	Coursework review
1.6	Software Engineering in the car industry (case study)	Requirements engineering session
1.7	Reading week	
1.8	Software design I (Architectural design)	Coursework review
1.9	Software design II (Architectural design)	Architectural design session
1.10	Software design I (System modeling)	Coursework review
1.11	Agile Software Development	Coursework review
1.12	Software Design for Security (case study)	Peer review assessment session
2.1	Software implementation I (Software Configuration Management)	Coursework review
2.2	Software implementation II (Code documentation)	Coursework version control setup
2.3	Software implementation III (Design Patterns)	Coursework code documentation
2.4	Open source Development	Design patterns talk shop
2.5	Software Engineering for game development	Peer review assessment session
2.6	Reading week	
2.7	Testing I (Development testing)	Peer review assessment session
2.8	Testing II (TDD and Automated Testing)	Exam preparation quiz
2.9	Testing II (Usability evaluation)	Usability evaluation talk shop
2.10	Software evolution and maintenance	Peer review assessment session
2.11	Software Engineering for mobile apps (case study)	Exam preparation quiz
2.12	Revision lecture	Mock exam

4) Cluster 4 (the second half of TB2) covered software testing and maintenance (weeks 2.7-2.10) (Table 2).

The end of each cluster was marked by lectures dedicated to discussing a Software Engineering related case study (weeks 1.6, 1,.11, 1.12, 2.5, 2.11).

The redesigned version of the course introduced five types of practicals: 1) coursework reviews, 2) peer review assessment sessions, 3) exam preparation sessions, 4) talk shops, and 5) SE task simulation sessions around key tasks in Software Engineering (requirements engineering, architectural design, code documentation). The coursework reviews practicals followed the template of the first two years, and were designed as discussions around each team's progress on the coursework. The talk shops sessions consisted of open discussions around SE topics not covered by the lectures, but theoretical in nature. The exam preparation sessions were designed as quizzes focused on key theoretical SE concepts, such as requirements, architectural patterns, testing strategies, or software development lifecycle models. The peer review assessment sessions and the SE task simulation sessions are presented in detail in Section 3.

3 CHANGES APPLIED TO THE COURSE

3.1 Realistic Deliverables (P1, P2)

The redesigned course aimed to replicate real-world software development scenarios by exposing students to the processes, methods, and tools software engineers use in their day-to-day work. Students were required to work in teams to develop a medium-sized software system of their choice. The cohort was divided into 38 teams of 5-6 students. The teams were decided by the course coordinator, and remained constant throughout the development process. All team members were encouraged to contribute to all steps in the development. There were no restrictions imposed in terms of methods or tools to be used, but the students had to ensure that the following six deliverables were completed and submitted on time (Table 3):

1) Project Proposal and Plan (PPP) describing the need for the system they decided to develop, and a detailed plan of the proposed development process.

2) System Requirements Specification Document (SRS) describing the methodology used for eliciting and specifying user and system requirements, and reflecting on the analysis of the user requirements leading to the set of system requirements.

3) Design Documentation (Design) translating the solution proposed for the system into design models. The teams could use any modelling tool or language, but had to include use case and sequence diagrams, architectural models, and user interface design models.

4) Prototype demo (Demo) consisting of a 3-5 minute video demonstration of the system. The working features of the system needed to be demonstrated, while the features not yet implemented needed to be demonstrated using story boards, or mock-ups.

5) Testing Documentation (Test) including a full description of the unit, component, and system testing of the system under development together with the description of the usability evaluation process used to evaluate the system.

6) Final Submission (Final) reflecting on the process followed, the decisions taken along the way, the issues encountered and the workarounds employed, and a critical analysis of the applied methods, tools, and processes. A demo of the system developed had to be submitted as part of the final submission. Additionally, the source code accompanying the system developed had to be made available on GitHub.

Methods, tools, and techniques for tackling common software development problems were introduced during the lectures. However, students were free to select which ones to apply in their work on the coursework. For example, in terms of version control management, the lecture described and demoed GitHub; some teams, however, decided to use other version control systems, such as BitBucket. Similarly, while the lectures described interviews, focus groups, ethnography as methods for requirements elicitation, teams were free to choose the method(s) they wanted to use for eliciting the requirements for the systems they were developing.

The deliverables were spread over the academic year, with monthly deadlines; all submissions were of group authorship. Students had the option of identifying individual contributions to the work, either by naming the authors of each section in the deliverable, or evaluating individual contributions in terms of percentages. Due to the specific nature of the deliverables, precise marking schemes defined for each ensured a higher level of consistency in marking (Table 3). Real-world examples were provided for each deliverable to expose students to the structure and content of such documents. Although their complexity was high, the examples helped students develop the analytical skills required for critically evaluating software documentation.

3.2 Software Engineering Tasks Simulations (P5)

Hands-on sessions around core course concepts, namely requirements engineering, architectural design, and code documentation, were developed to simulate typical Software Engineering tasks. Each session lasted for one hour. The cohort was divided into 10 groups, and each session was organised separately for each group. Below, we describe these sessions in more detail.

3.2.1 Requirements Engineering. The practical session on Requirements Engineering (RE) addressed the misconception that RE is a limited exercise in describing a minimal system aim, broadly describing system features, and hinting at elements of a system's user interfaces. We designed a role-playing exercise to give students a short, sharp introduction to the different phases of RE. Students acted as developers for a mobile app start-up company. Their investors see an opportunity for developing a diary app for a very influential, but very secretive group of users. These users were *extreme characters* [3], which exhibited hidden character traits, and added an extra dimension to the challenge of eliciting requirements that meet their expectations. As part of the development team, they were given access to representative users. The exercise was further organised into four steps.

1) Familiarisation. Students were divided into two groups, A and B. Each group was briefed on the target end-user they represented. Acting as their respective target end-users, namely *the Queen* and *a double agent*, the groups agreed on the functionality they expected from the app. Students were forbidden from revealing their role outside the group to avoid issues with bias, i.e. students in Group A thinking they implicitly understood Group B extreme character user requirements.

2) Elicitation. Each group was further divided into 2 sub-groups: A1 and B2 were the interviewers and A2 and B1 were the interviewees. Interviewers acted as requirements analysts, while interviewees acted as the user they represented based on the brief. Group A1 interviewed group B1, while group B2 interviewed group A2. Instructions on interviewing were provided to all groups.

3) Specification. Students returned to their initial two groups. Each group specified the requirements elicited during the previous step. Students could use any requirements specification technique they felt was best suited, but the specification needed to be agreed by the whole team.

4) Validation. Students returned to the groups formed for the Elicitation step, namely A1, A2, B1, B2. This time, A2 and B1 acted as interviewers and A1 and B2 acted as interviewees. Interviewers acted as requirements analysts validating the specified requirements, while interviewees acted as the user they represented. Group A2 interviewed group B2, while group B1 interviewed group A1.

We describe the session in detail together with its evaluation in [6].

3.2.2 Architectural Design. Architectural design in Software Engineering proved to be one of the most challenging topics. Students had previously only developed small programs, web sites, or simple mobile apps. The need for designing and documenting the architecture of such systems based on guidelines and design patterns was seen as an unnecessary complication. The benefits of reasoning around software architectural design are only made apparent when (re)engineering a complex, large-scale system. However, such systems are not easily accessible and fully understanding how they work and how they are being developed requires more time and resources that an introductory course in Software Engineering can afford. We addressed this problem by providing students with a high level description of an air traffic control (ATC) system [2] explaining the goals of the system, the type of information it needs to operate, the types of outputs it produces, its main software components and the ways these components communicate, and a graphical representation of the architectural model of the system. Four categories of problems were described for the system in its current architecture; some of these problems had simple solutions, others needed a complete redesign of the system and its architecture. All problems were described in detail, and examples of potential consequences provided. The exercise was designed around three stages.

1) Recognition. Students were asked to recognise the architectural pattern used in designing the system's architecture. Working in pairs, they discussed the relationship between the problems identified for the system and its architecture, with the aim of identifying those elements of the architectural model leading to or exacerbating the problems identified.

2) Creation. Working in pairs, students chose a different architectural pattern from a list, and redesigned the ATC system's architecture to follow that pattern. Students were provided with a brief of the pattern chosen, and comparative discussion of the two patterns was encouraged.

Table 3: Marking scheme for all coursework deliverables: Deliverable, Task to complete as part of the deliverable, Percentage contribution to coursework mark

Deliverable	Task	Mark
PPP	Describe project organisation (1%) Produce risk analysis (1%) Identify the resource requirements for the project (1%) Breakdown work into tasks (1%) Schedule project (1%)	5%
SRS	Describe the need for the system and the context of its development (2%) Describe the process used for eliciting requirements (5%) Provide an overview of the high-level user requirements for the system (3%) Document the functional (user and system) requirements of the system (5%) Document 3 types of non-functional requirements of the system (5%)	20%
Design	Provide use case modelling for at least 5 scenarios (6%) Provide sequence diagram modelling for at least 5 scenarios (6%) Provide system architecture: representation and architectural patterns used (6%) Provide at least 4 mockups for the system's UI (2%)	20%
Demo	Provide an overview of the prototype's capabilities (2%) Walk through the prototype's capabilities (6%) Provide voice-over/annotation highlights (2%)	10%
Test	Provide details on the design of each of the testing phases: unit, component, and system testing (5%) Provide details on the testing process followed: set up, testing values, issues encountered, tools used (5%) Describe the results of the testing process (5%) Provide details on the process followed for evaluating the usability of the system (5%)	20%
Final	Make source code available on GitHub together with running instructions document (8%) Provide a demo of your final product: walkthrough of all capabilities of the system, voice-over/annotations (5%) Provide evidence of system documentation (2%) Provide a retrospective account of the project (10%)	25%

3) Critique. Students swapped the architectural model they redesigned for the ATC system with another pair, and critically evaluated each other's results. They were asked to decide the extent the architectural model reviewed addresses the issues identified for the system.

The session exposed students to an interesting but complex, large-scale system they were unfamiliar with, provided the rationale for reasoning about software architectural design, and helped them develop critical skills in making sense of and reengineering existing architectural models.

3.2.3 Code Documentation. Motivating students to document their code is difficult for several reasons. First, when coding, students are focused on getting the code right at that moment, and lack a long term view of how the code they write will be used by their peers in the future. Second, being exposed only to small, low-complexity systems, students fail to see the rationale behind making their code understandable to others. Finally, when they do comment their code, students rarely use best practice guidelines for improving the quality of those comments. To address these problems, we created an exercise asking students to clone an existing medium-sized open source software system with all its comments removed, document the code, and submit the changes made to the code back to the version control system. Students were issued with a set of guidelines on how to write high quality code comments. The exercise brief was made available online, and a deadline for the code contributions to be submitted to the version control system was set. Students could complete the exercise anytime before the deadline, and the most relevant and meaningful comments were then merged into the main repository.

3.3 Case Study Exercises (P4, P5)

As discussed in Section 1, one of the main challenges in teaching Software Engineering to undergraduates is making the need for software processes and documentation obvious. We addressed this by including open discussions around Software Engineering case studies. The case studies discussed were of two types:

1) Consequential: aiming to identify the consequences of not following a process when developing software, not documenting the simplified process followed, and not involving all the stakeholders in the process, eg. [10]. The discussion focused on the costs of overlooking the process in Software Engineering and the necessity of method in developing complex software systems.

2) Exploratory: portraying the specifics of Software Engineering in various industries, e.g. the car industry, the video game industry, and the mobile apps industry. The discussion focused on identifying, for each SE tasks - i.e. planning, analysis, design, implementation, testing, maintenance - the elements specific to the particular industry, the ways in which they compare to other industries, and the interplay between other disciplines and Software Engineering.

Using these two types of case studies had two benefits. First, the *consequential* case studies illustrated the context around the software being developed, identifying the implications of "writing code" to the practice of Software Engineering. Second, Software Engineering today takes many shapes and forms; therefore, the *exploratory* case studies developed a more holistic understanding of the types of methods, tools, and processes software engineers use to develop a diverse range of software.

3.4 Software Engineering Tool Demos (P4)

Ensuring students can confidently use software development support tools is vital. However, that is difficult to achieve without a trial and error process on the students' part, and that is difficult to run for a large cohort. To address this issue, we provided students with video tool demos, allowing them to rewind and replicate them in their own time. We looked at four classes of tools: a) Configuration management and version control (submitting a pull request in GitHub, committing changes to a repository, cloning an existing repository, managing conflictual commits); b) Source code documentation (generating a project's documentation and browsing an existing project's documentation); c) Unit testing (setting up a test case, and running a test suite for a project); and d) Continous integration (TravisCI-GitHub integration capabilities). All video demos were made publicly available, and incorporated in the teaching material presented during lectures and tutorials.

3.5 Peer-review Assessments (P3, P5)

As a tool consistently used in all aspects of Software Engineering, we incorporated peer-reviews in the redesign of the course. For each submission described in Section 3.1 (except for the Final Submission), a one-hour practical session was devoted to its peer-review. Each team was provided with the deliverable submitted by another team, and all teams were provided with an assessment scheme to guide the review process. The assessment scheme was based on the marking scheme used in the formal assessment process, but was augmented to encourage discussion. The lecturer running the peer-review sessions acted as moderator, asking probing questions when needed, and steering the conversation to areas not considered by the students. Students had the option to engage in a dialogue with the teams whose work they were reviewing, and all took up this opportunity.

For the first submission, teams were reluctant to express any criticism, and tended to provide generic comments on the work reviewed (eg. "this is good", "it covers most points"). However, engagement increased during later submissions, with students refining and deepening their criticism for the work under review. For example, during the Design Documentation peer-review sessions, some students required evidence for component selection decisions, and commented on the justification for selecting particular architectural styles based on the number of users expected. In addition to the peer-review sessions, all teams were provided with a breakdown of each submission's mark together with the lecturer's comments on how the submission was judged against each marking criterion. The feedback was mostly a summary of the points discussed during the peer review session, with an emphasis on the take-home points for each submission.

3.6 Drop-in Mediation Sessions (P6)

In addition to the lectures and practicals described in Table 2, drop-in mediation sessions were scheduled as weekly two hour dedicated sessions to be used by teams when reaching a state of conflict or deadlock. The sessions were informal and required all members of the team to be present. Some of the issues encountered included: a member of the team not meeting deadlines, or not attending meetings, teams not being able to reach a definitive decision on parts of the development process, or teams not reaching agreement on the division of work.

4 RESULTS

We compared the student feedback and student performance for the course for the year it underwent redesign (2016/2017) and the two previous years (2015/2016 and 2014/2015).

Student Feedback. All student feedback was provided at the end of the academic year, with students providing both qualitative and quantitative feedback. In terms of qualitative feedback, students report on the aspects they mostly enjoyed about the course, and on the aspects of the course needing improvement. In terms of quantitative feedback, students were asked to rate their level of agreement on a scale from 1 (strongly disagree) to 5 (strongly agree) with the following statements:

S1: The course makes a positive contribution to my overall course;

S2: I am clear about what I need to do to be successful in this course;

S3: Lecturers are good at explaining things on this course;

S4: Lecturers use of the Virtual Learning Environment (VLE) helped me to learn;

S5: I am able to communicate with lecturers teaching on this course when I need to;

S6: The workload for this course is manageable;

S7: Assessment arrangements and marking criteria are fair;

S8: I have had opportunities to get feedback on my work during this course;

S9: Feedback on my work during this course helps me clarify things I do not understand.

The structure of the survey is unchanged across the three years we compared. The number of respondents varied throughout the years, with 44 (23%) answers for 2016/2017. This quantitative feedback is summarised in Figure 1.

Assessing the course structure and contribution focused on five themes: a) VLE (S4), b) clarity of the assessment process and course goals (S2), c) student-lecturer communication (S5), d) clarity of the teaching process (S3), and e) course contribution (S1). The redesigned course scored higher on all aspects. Several changes contributed to these results, including: a) incorporating demos of SE tools in the teaching material, b) defining focused assignments specifications, and providing clear marking schemes for each of the assignments, and c) incorporating hands-on practical sessions and case study discussions. On the topic of the coursework workload (S6), the redesigned course scored lower. We expected that, giving the students the freedom to work on the project of their choice, they would find the work more manageable. However, due to the extent of the changes incorporated in the redesign of the coursework (i.e.

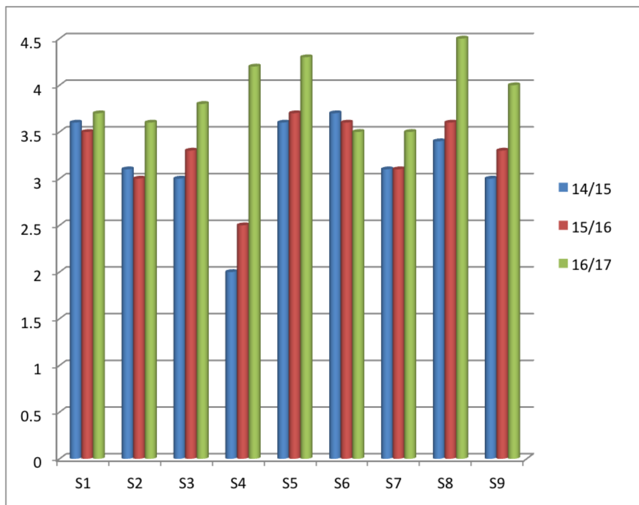

Figure 1: Feedback on course contribution (S1),communication (S2-S5), workload (S6), work feedback (S7-S9),

students were required to work as a team throughout the year and deliver evidence of the progress of their work monthly), this was not the case.

Student feedback on the feedback students received for their work was covered by three of the questions in the survey, with the focus on the marking criteria (S7), opportunities to get feedback on work (S8), and the quality of feedback received (S9). The redesigned version of the course scored higher than the previous year on all aspects. The peer-review assessment contributed significantly to these results, as these were seen as an opportunity to get feedback on their work from both their peers and the lecturer. Moreover, given the informal aspect of these sessions, students did not feel constrained or intimidated from asking their peers for clarification or debating some of the points they did not agree with.

Student Performance. No significant difference was noticed in terms of the mean of the marks and standard deviation (Table 1). We are satisfied with this performance given the significance of the redesign and the increase in the cohort size.

5 DISCUSSION AND LESSONS LEARNED

This section discusses how the changes were perceived by students, and how they addressed the challenges depicted for the course.

Ensuring consistency in assessing coursework and marks. The mark assigned for the coursework was worth 50% of the final mark for the course. Some students believed this was unfair given the teamwork element of the coursework, and given the imposition of teams (*"Group coursework worth 50% is unfair when groups are randomly chosen"*). The suggestion was to design a smaller-scale teamwork element as part of the coursework, and assign this a lower weight. Being given the opportunity to assess other teams's work and have their work assessed by others was particularly appreciated by students. This gave them assurance that all work was marked consistently and transparently, across teams.

Managing teamwork for a large number of teams. Students enjoyed the freedom of choosing and customising the systems they developed and the process used. They also appreciated working as part of a team, and trying to decipher each teammate's interests, and strengths & weaknesses (*"Developing a product by working with others by focusing on strengths and weaknesses"*). Students found the team sizes described in Section 3.1 to be too big to be efficient, suggesting a maximum team size of 4 students (*"This course should have groups of 4 or 3 as less people will develop a better quality of work and ensure everyone has equal say and role to produce better results plus be able to have everyone work in a group"*). Students were encouraged to be proactive in managing the team; instead of designating one project manager across the project, they would take turns in acting as project managers for various phases in the development process. However, for large groups this did not always work, with students feeling that getting 6 students to organise in the absence of a clearly designated team leader was too great a challenge (*"Our group for the coursework is too big (6 people). Without a project manager it makes it very difficult to organise everyone"*).

All teams were formed by taking into consideration other courses students were enrolled on, with the aim of having a diverse representation of skills and knowledge. Additionally, the aim of keeping the process as realistic as possible was achieved by assigning students to teams, and not allowing students to form teams themselves. This, however, was a source of discontent among students, with many preferring to work in teams decided by themselves (*"Groups should be chosen for group coursework by us as some teams do not work well"*, *"Unfair mix of people. Maybe consider some element of allowing students to pick team members"*), leading in some cases to an overall lack of enthusiasm for the course as a whole (*"Choose groups for work. Groups tend to be unenthusiastic because of this"*). The sentiment was that teams would have performed better had the option of choosing their peers been available (*"Choosing our own groups for coursework, I probably would have achieved better. I know it is similar to real life scenarios, but those who don't work wouldn't be in a job"*). Students felt that not all of their peers were equally motivated to contribute to the coursework and, in those instances, the team was at an disadvantage without an obvious and immediate solution (*"No choice of teams, teams with known issues with individuals at an immediate disadvantage"*)).

Students felt that more intervention from the course coordinator on handling out-of-the-ordinary situations was needed, with mediation considered too mild to make a significant difference (*"Better solutions for when group mates don't do work"*). Students felt the need for an authority to decide on issues they struggled to find consensus on, or to intervene when team members were not actively contributing (*"Potential better handling and monitoring of potential group work coursework or tasks"*). The expectation for the course coordinator to intervene in situations where the team was not working well were not entirely met based on the student feedback (*"Teams should not be forced upon. If it must then there should be some reassurance available or some fall back option where if nobody contributes, you don't fail along with them. Bad idea!"*).

Ensure student engagement and interactivity. Students found the redesigned course well structured, with the lectures covering topics relevant to their course and career interests (*"The course is very well structured, the syllabus makes sense regarding what we*

learn"). The interplay between practical case study discussions and theoretical topics was particularly appreciated. The structure of the coursework and the schedule for all deliverables worked well for the students. Although the coursework workload increased significantly, spreading the deliverables over the whole year appeared to make the workload manageable (*"Coursework is well spread over the year"*). With monthly deliverables, students were given enough time to work on each submission. At the same time, students kept engaged with the course, avoiding the situation where a large deliverable is due only at the end of the year (*"Work is due in chunks rather than all at once"*).

Providing all students with comprehensive feedback. The peer-review sessions were introduced not only as a means of formative assessment, but also as a vehicle for developing critical thinking for software documentation and code. The student feedback on these sessions was positive, with the feedback and session frequency considered helpful (*"Frequent feedback which helps me know where I need to improve"* or *"Seminars are very useful, especially the coursework feedback sessions"*). Many students suggested scheduling these sessions prior to deadlines, to give students the opportunity to incorporate the comments received into their submissions. The sessions also engendered a community feeling within the teams; this corroborates with previous work examining peer testing practices in Software Engineering projects [1], although that study considered the review of code-level artifacts only, rather than the broader range of Software Engineering artifacts reviewed for this course.

Supporting students in developing the skills required in SE. The re-designed lectures focused less on theoretical aspects of Software Engineering and more on real-world practices, process and tools used by professionals, and the characteristics of Software Engineering in various industries, such as game development, mobile app engineering, and the car industry. This exposure to the real world was valued by the student as good preparation for their future careers (*"Tie-ins to real world problems and case studies to prepare us for after university"*, *"I am able to learn new and interesting things which will help me in career aspects"*).

Students found the interactive sessions described in Section 3.2 helpful by making it easier to grasp to concepts and apply them to their work (*"The practical lessons are useful and helped me to learn a lot more easily"*, *"Tutorials are useful for examples of areas covered in lectures"*). They also felt that the they were designed as realistic, meaningful exercises (*"Lectures and practicals are meaningful and helpful"*). Students also appreciated putting into practice the concepts discussed during lectures and tutorials, and using their coursework as a test bed (*"Understanding the different methods of implementing plans and requirements into a current project"*). Due to the collaborative nature of the coursework, students developed the soft skills required for working as part of a team (*"Also it has improved my communication skills with other students"*).

6 RELATED WORK

Related work in the teaching of Software Engineering mostly focuses either on specific educational aspects, such as the design of team projects, peer-review exercises, and the role of educational methods such as "the flipped classroom" [12], or on teaching focused topics in Software Engineering, such as software architecture or requirements engineering [9]. There are few studies describing year-long experiences running programming and object-oriented courses, in situ changes, and their impact on student experience and performance, and guidelines on managing the challenges posed by teaching programming as part of the undergraduate curriculum [7, 8].

Robillard [13] takes a practical approach to teaching Software Engineering by designing the unit as a project course with teachers acting as supervisors or clients. Paez [12] use the flipped classroom approach with good results for small size classrooms (i.e. 20 students), but no evidence is provided for using the approach on a large cohort. Nordio et al [11] describe some of the scaleability challenges teaching software engineering where distributed software development is used as a case study; their course was designed around four phases of development, namely requirements, interface specification, implementation, and testing. The course design was used across 4 years, with annual increases in student performance. Garousi at al. [4] note that peer review exercises are rarely used in educational contexts and, when used, they tend to be limited to engineering courses. The authors describe their experience with using such exercises as part of a software engineering course where students were asked to find defects in their peers' code submissions. The accuracy of the peer reviews was determined by the instructor. Based on this trial, the authors ask for more evidence of peer review usage in education (particularly with large classes), emphasising the advantages and the challenges they bring. Gehringer [5] describe Peer Grader, a system designed to support peer review exercises for students. The tool facilitates a semi-automatic peer review process where students are able to review and grade each others work.

7 LIMITATIONS AND FUTURE WORK

The redesign of the course comes with a set of limitations. First, we did not put in place a mechanism for monitoring and, possibly, predicting the time allocated to solving conflicts within teams for both the students and the course coordinator. Based on our experience running this course, we expect such considerations to make a positive impact on the overall team experience. Logging all reported conflicts and documenting the mediation strategies and resolution procedures for all of them could, potentially, lead to a valuable resource that can be shared across future cohorts.

Second, the redesign does not formally define a process for documenting the consequences of conflicts within teams on the overall performance of the teams. Such follow-up strategies could help better predict the outcome of various types of conflicts, and allow early intervention by the course coordinator. This can prove particularly helpful when working with large cohorts as it would help the course coordinator prioritise issues and their resolution.

8 CONCLUSIONS

In this paper, we present a study on the redesign of a year-long Software Engineering course for a large cohort. Based on our results, we recommend a number of changes that have been positively received by our students. These include: keeping students in the loop throughout assessment and marking by having them

peer-review each others' work, setting up mediation sessions for managing teamwork dynamics, providing students with the freedom of choice with respect to the process and tools they use, but ensuring focused assignments with precise marking schemes. Resource management needs to be reconsidered for large cohorts to support playback and independent trial and error exercises. Building the much needed software engineering skills for a large cohort requires scalable exercises able to simulate real world software engineering tasks. Taking inspiration from Interaction Design [3] and Software Architecture [2], we have designed such exercises to teach requirements engineering, architectural design, and code documentation.

REFERENCES

[1] Nicole Clark. 2004. Peer Testing in Software Engineering Projects. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30 (ACE '04)*. Australian Computer Society, Inc., 41–48.

[2] Jackson D. and J. Chapin. 2000. Redesigning air traffic control: an exercise in software design. *IEEE Software* 17 (2000). Issue 3.

[3] J. P. Djajadiningrat, W. W. Gaver, and J. W. Fres. 2000. Interaction relabelling and extreme characters: methods for exploring aesthetic interactions. In *Proceedings of the 3rd conference on Designing Interactive Systems*. ACM, 66–71.

[4] Vahid Garousi. 2010. Applying Peer Reviews in Software Engineering Education: An Experiment and Lessons Learned. *IEEE Transactions on Education* 53, 2 (2010).

[5] Edward F. Gehringer. 2001. Electronic Peer Review and Peer Grading in Computer-Science Courses. *SIGCSE technical symposium on Computer Science Education* 33, 1 (2001).

[6] Claudia Iacob and Shamal Faily. 2017. Using Extreme Characters to Teach Requirements Engineering. In *Proceedings of 30th IEEE Conference on Software Engineering, Education, and Training*. IEEE, 107–111.

[7] Erkki Kaila, Einari Kurvinen, Erno Lokkila, and Mikko-Jussi Laakso. 2016. Redesigning an Object-Oriented Programming Course. *ACM Transactions on Computing Education* 16, 4 (2016).

[8] Linda Marshall, Vreda Pieterse, Lisa Thompson, and Dina Venter. 2016. Exploration of Participation in Student Software Engineering Teams. *ACM Transactions on Computing Education* 16, 2 (2016).

[9] Sandeep Mitra. 2014. Using UML Modeling to Facilitate Three-Tier Architecture Projects in Software Engineering Courses. *ACM Transactions on Computing Education* 14, 3 (2014).

[10] Erich Musick. 2006. The 1992 London Ambulance Service Computer Aided Dispatch System Failure. *Formal Methods* (2006).

[11] Martin Nordio, Carlo Ghezzi, Bertrand Meyer, Elisabetta Di Nitto, Giordano Tamburrelli, Julian Tschannen, Nazareno Aguirre, and Vidya Kulkarni. 2011. Teaching software engineering using globally distributed projects: the DOSE course. *Community Building Workshop on Collaborative Teaching of Globally Distributed Software Development* (2011).

[12] Nicolás Martín Paez. 2017. A Flipped Classroom Experience Teaching Software Engineering. *IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials* (2017).

[13] Pierre N. Robillard. 1996. Teaching Software Engineering through a Project-Oriented Course. *9th Conference on Software Engineering Education* (1996).