Teaching Software Engineering with Free Open Source Software Development: An Experience Report

Anya Tafliovich
University of Toronto Scarborough
atafliovich@utsc.utoronto.ca

Thomas Caswell
matplotlib
Brookhaven National Laboratory
tcaswell@bnl.gov

Francisco Estrada
University of Toronto Scarborough
festrada@utsc.utoronto.ca

Abstract

We report on the design and delivery of a senior Software Engineering course within the limits of a Computer Science program. The course is structured around a collaboration with a large, active Free Open Source Software project. We show how this structure allows us to (a) incorporate principles of Project Based Learning and of Service Learning, reaping the benefits of these pedagogies, (b) effectively, using a hands-on approach, teach a number of essential topics in Software Engineering, (c) provide the students with a capstone project experience, given the lack of one in our curriculum, and (d) use the project as a powerful motivating factor for the students.

We outline the experiences of the course instructor, of the teaching assistants team, and of the students of the course. We also describe the experience of the lead developers of this open source project, and report on the benefits and costs (time commitment) to the project.

1. Introduction

Much has been said about the gap between the industry expectations and the needs and abilities of recent Computer Science graduates [1, 2]. Promptly after graduation many students are required to dive into a code base that is orders of magnitude larger and more complex than any projects they have worked on through their university courses. Importantly, this code base is not an academic project, or an ad-hoc, course-based distribution designed with specific learning objectives in mind. Instead, it is fairly often a poorly documented and sometimes a poorly designed system [3, 4], which is difficult to properly maintain even by a seasoned developer. To be prepared to enter the workforce, students need a solid foundation in Software Processes, Software Design, Software Verification and Validation, Software Quality, Security — the list continues.

Producing quality Software Engineers within a Computer Science (as opposed to Engineering) department/faculty, presents additional challenges. A Computer Science program that is solid, rich, and well-grounded in Mathematics and Theoretical Computer Science leaves little room for the applied, hands-on courses that teach students how to build and maintain well-designed, high quality software, how to document and test it, how to use state-of-the-art industry processes and tools, how to work in a team and communicate effectively, etc.

In recent years, Computer Science and Software Engineering educators are becoming increasingly interested in Project Based Learning and, not necessarily related, in Service Learning (see Section 2). Many institutions now require a capstone project course or courses. These senior-level courses provide students with an opportunity to integrate and synthesise the knowledge learned in multiple courses and to apply it to a real problem. By definition, a capstone project must constitute an authentic, project-based activity that closely relates to professional work in the field. Students must apply the discipline knowledge and skills acquired in their program, as well as generic skills, such as communication and teamwork skills. The benefits of having a capstone project in the program curriculum have been well reported and recognized (see Section 2).

In this paper, we will describe our experience developing and running a senior-level Software Engineering course within the limits of a Computer Science program. The course is designed and structured around a collaboration with `matplotlib`, a large, active, widely used Free Open Source Software project. This collaboration:

- incorporates principles of Project Based Learning and of Service Learning, reaping the benefits of these pedagogies,
- allows us to effectively, using a hands-on approach, teach the following topics:
  - software modelling and analysis,
  - software architecture and design patterns on a large scale,

URI: https://hdl.handle.net/10125/60211
ISBN: 978-0-9981331-2-6

  - software quality analysis,
  - verification and validation,
  - technical writing,
  - agile software development (reinforce),
  - project planning and estimation,
  - teamwork and soft skills, and
  - professional practice,

- provides the students with a capstone project experience, given the lack of a capstone project course in our curriculum, and

- serves as a powerful motivating factor for the students.

We outline the experiences of the course instructor, of the teaching assistants team, and of the students of the course. We also describe the experience of the lead developers of this open source project, and report on the benefits and costs (time commitment) to the project.

2. Related Work

Over the past two decades, Project Based Learning [5] has been gaining acceptance as higher education continues to shift toward student-centred learning. Although a vast majority of available literature on Project Based Learning is devoted to domains outside of exact sciences and information technology, interest in using Project Based Learning in Computer Science classes is growing. Recent work by Putcher and Lehner [6] reviews over five hundred instances of incorporating Project Based Learning in Computer Science classes. The authors report on the benefits of the approach, as well as identify critical success factors for such projects.

In recent years many universities have introduced a capstone project in the curriculum. Capstone projects enable students to integrate and synthesise the knowledge learned in multiple courses and to apply it to a real problem — an essential part of preparing students for the workforce. The work of Dugan [7] provides an excellent overview of the available capstone projects literature, categorising the course models and course topics, and summarising the benefits of the courses. The work of Levia [8] studies an important aspect of a capstone project — its evaluation, and suggests grading rubrics that both encourage the students to remain engaged in the project and enable the instructor to diagnose student learning. The work of Todd and Magleby [9] examines success factors of capstone projects, with an emphasis on the needs of all stakeholders, not just the students.

The work of Pinto *et. al.* [10] looks at introducing FOSS projects in SE courses from the professor's perspective. Our experience aligns with some of the reported benefits to the students, such as improved students' communication and technical skills and the importance (as perceived by both the students and the employers) of enhancing the students' resumes. We also experienced some of the challenges reported in their work, such as significant demands on professors' effort and time, as well as a challenge and importance of creating appropriate assessment schemes. In the following Sections we describe our assessment scheme, which evolved over several offerings of our course, and which we believe to be appropriate for our setting. Another reported challenge, namely the required fast response on the part of the FOSS project lead developers, we addressed by forming a long-term effective partnership with the project lead developers (one of the authors of this work is a lead developer for the project). The demands on the professor's effort and time is still something we have not found effective ways to address, and this remains to be a challenge in our experience as well.

In recent years we have seen growing interest in Service Learning, both at the high-school level and in higher education. It has been shown that Service Learning not only increases learning of the subject [11], but also influences students' personal and social development [12], and has far reaching academic, personal, social, and citizenship outcomes [13, 14]. There is also literature that examines success factors of Service Learning projects, e.g. [15] and [16].

The work of Jamieson [17] examines the application of Service Learning specifically to Computer Science and Engineering Education. The work of Webster and Mirielli [18] examines the students' perspective on this topic. Interestingly, recent work of Dahlberg *et. al.* [19] suggests that Service Learning may help us attract and engage under-represented students — a problem faced by every higher education institution teaching Computer Science or Engineering.

3. Environment

The course we describe here is offered at a large (over $60,000$ undergraduate students between multiple campuses), public, research-intensive North American University. Ours is not an Engineering Department / Faculty, and the undergraduate programs we offer are in Computer Science, rather than in Software Engineering. Students choose to specialise in one of several "streams", and Software Engineering is one such stream. Approximately 80% of Computer Science

students opt for the Software Engineering stream — likely because of the promise of better employment opportunities upon graduation.

The length of a term in our institution is 12 weeks.

3.1. Software Engineering Stream

As we mention above, our SE stream is heavily theoretical. The program prescribes 13.5 credits out of the 20 credits required by the University for graduation. Of these 13.5 credits, the following required courses are what we would label as "applied", or "programming", or "software development" courses:

1. CS1 and CS2 are fairly typical introductory courses that students take in their first year; these are taught in Python, in an object-late fashion,

2. a second-year course, which introduces basics of object-oriented design, some tools for effective collaboration and version control, and provides the first opportunity to practice working in teams; the course is taught in Java and sometimes includes Android,

3. a second-year Computer Architecture course, which introduces boolean algebra, digital circuits, and assembly language,

4. a second-year Introduction to Systems Programming taught in C,

5. third-year Introduction to Databases, Introduction to Operating Systems, and Principles of Programming Languages courses,

6. a third-year Introduction to Software Engineering course, which serves as a direct prerequisite for the course we describe in this report and, finally,

7. the course itself.

A keen reader will notice, in particular, the lack of a capstone project in the curriculum.

Examining the above list, we see that the subject of Software Engineering in the Software Engineering stream essentially needs to be covered within two 12-week-long senior courses.

Following the general North American trend, the enrolments in the course have steadily increased from 32 students in 2012 to over 160 students in 2018. An overwhelming majority of the students registered in the course are in their final year of the program.

3.2. Software Engineering Courses

The 2015 edition of the ACM and IEEE Computer Society Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering [20] lists, among others, the following areas of knowledge: Professional Practice, Software Modelling and Analysis, Requirements Analysis and Specification, Software Design, Software Verification and Validation, Software Process, Software Quality, and Security.

As we describe above (Section 3.1), we are faced with the challenge of introducing these topics within the scope of two senior courses. Our approach, briefly and at a very high level, is as follows.

The first (third-year) Software Engineering course focuses on *small-scale systems*. In this course students learn about Requirements Elicitation and Analysis, Project Management and Planning, Object-Oriented Design and Design Patterns, basic Software Modelling, version control with git, and elements of Verification and Validation. The course is taught with a strong emphasis on Agile Software Development / Scrum, and students are required to use Agile Software Development in their project. The project is a team project, it is 9–10 weeks long, and involves eliciting requirements from a real client and designing, developing, and validating a software product for the client.

The second (and last!) Software Engineering Course looks at *large scale systems*. This is where our partnership with Free Open Source Software community of `matplotlib` becomes invaluable.

For the vast majority of the students this is their very first experience working with a large code base. The reader should realise what a difficult task this is for an inexperienced student. The team is given over 200,000 lines of code in hundreds of files, a requirement for a new feature (or to fix something that is broken), and a time limit. They need to figure out how to make this happen! The vast majority of the students, at the beginning of the course, have no idea where to even start looking within this large amount of code.

3.3. The Co-op Stream

Our institution also offers a Co-op Program in Computer Science. The program includes three mandatory co-op terms (internships) four-month-long each. The program is very competitive: in addition to the higher admission requirements, the students are also required to maintain their GPA in order to stay in the program. The numbers vary from year to year, but on average only 30% of our students are enrolled in it. Higher tuition may also be a deterring factor.

4. Learning Objectives

We had several sources of inspiration when designing the course. In addition to the ACM and IEEE Computer Society Curriculum Guidelines, we referred to the work of Radermacher *et. al.* [21] in aiming to bridge the gap between industry needs and expectations and typical abilities of graduates. As a result, we made every attempt to provide the students with an opportunity to learn about and to practice:

- software modelling and analysis,
- software architecture and design patterns on a large scale,
- software quality analysis,
- verification and validation,
- technical writing,
- agile software development (reinforce),
- project planning and estimation,
- teamwork and soft skills,
- professional practice,

and last, but not least, to have a capstone project experience, given the lack of a capstone project course in our curriculum.

In addition, we had another important objective: we wanted to *motivate* the students. As in many other institutions, we have witnessed a general decline in the levels of and a qualitative change in the kind (from internal to external) of student motivation [22].

5. Towards Achieving the Learning Objectives

In this Section we describe how incorporating a Free Open Source Software project into the course facilitates achieving the above goals.

Because of the curriculum restrictions, we are unable to cover the knowledge areas in class prior to the commencement of the project. Furthermore, it has been suggested that this is not the best way to structure student learning [23]. We therefore opt for Just-In-Time teaching. In other words, we cover a topic in class / lab immediately prior to when the students begin to incorporate this knowledge in their project work.

Keeping in mind that by the end of 12 weeks student teams should have been provided with an opportunity to make a *significant contribution* to a large open-source project, we structure the course as follows.

Students work in small size teams with four or five students in each team. Of course, as some students choose to withdraw from the course in the early weeks of the term, we are sometimes forced to restructure. As a rule, we decided to leave groups of four as is, and to merge and/or rearrange groups of three or fewer students. Forming these teams is a non-trivial task to say the least. In [24] we discussed strategies for effective student team formation and management. In addition, as we mention in Section 3.3, approximately 30% of our students are in the Co-op program. Having at least one student from this program in each team provides benefits for both the Co-op students, as they have an opportunity to mentor their peers in the use of tools they may already be familiar with from their internship experiences, and to the rest of the team, who have immediate access to extra help.

As the term is only 12 weeks long, the length of the project is 11 weeks. The students are required to work on it continuously throughout the term: last-minute-cramming is discouraged as marks are deducted. Each team has a teaching assistant assigned to guide the team through the term, as well as to evaluate the process, team's progress, and the final product. To ensure continuous help and feedback, and to monitor and steer the agile development process, the teams meet with their TA on a weekly basis. As an additional benefit, this structure enables several instances of written and oral presentation, which provides a badly needed opportunity to repeatedly practice and get feedback on these important skills, largely neglected in the rest of our curriculum.

Furthermore, the teams are given an opportunity to address the TA's feedback and to re-submit their work for re-evaluation (which has a small, but non-negligible effect on the grade for that phase). We found this provides a strong incentive for studying the TA's feedback, and understanding and correcting the errors, thus improving their chances of future success.

Finally, evaluation of the project is a non-trivial course design decision. While educators uniformly agree on the importance of using student team projects, there is no agreement on best methods of forming and managing student teams and on best policies of evaluating student team projects [25]. As a result of an extensive literature overview (see Section 2), as well as our own results on evaluating team projects [26, 27], we developed the following method of evaluation.

Having the project divided into several phases / deliverables provides us with an opportunity to offer multiple instances of evaluation — both the

evaluation of the teams' work by the teaching assistant and the evaluation of individual's work by her peer teammates. Peer evaluation is done anonymously (to the students, not to the instructor), and the students are presented with an aggregate "score" that reflects their peers' beliefs on their individual performance. The individual grade is then obtained from the team grade by adjusting it according to this score.

The first rounds of both TA and peer evaluations are performed early on in the term, providing the students with an opportunity to learn if they have not been performing up to the team's standards, and to correct their behaviour. It is important to note that team members were not evaluated on their general knowledge or ability to write good code, etc., but rather on the quality of their collaboration with their teammates.

Finally, every contribution accepted into the code base of the open source project is additionally rewarded with marks.

5.1. Learning from the Code Base: Reverse Engineering and Software Modelling

The first major deliverable in the student project involves learning the code base. Students have three weeks to work on this deliverable.

The teams begin by forking the project, installing it, reading the documentation, and learning to use the tool. A more important and a more challenging task is learning and analysing its software architecture.

The deliverable requires the teams to provide a commentary on the architecture of the system, highlighting interesting aspects of the design (e.g., architectural style, degree of coupling, etc.), discussing the quality of the architecture used in the system, and suggesting possible improvements where appropriate.

Teams are required to submit a detailed description of the overall architecture of the system, using UML diagrams to illustrate the points they wish to make. Students have a choice of notation they deem appropriate for their purpose (e.g., packages, components, interfaces, etc.), and, in general, the exact UML notation that they use is much less important than the modelling decisions they make: what is important enough to include, what to omit, how to structure the diagrams, etc. They need to show clearly where the classes belong in this architecture, and what external packages the system interacts with.

Students use reverse engineering tools to generate UML diagrams representing the modules, classes, subclass relationships, and associations in the source code. The majority of the modelling work is in editing the generated model to capture information missed by the tool, to remove unnecessary detail, etc., in order to provide a clear picture of the system architecture.

Corresponding to the tasks in this deliverable, the contents of the preceding lectures includes Software Modelling and Analysis, Software Architecture and Design Patterns on a large scale, and Software Quality Analysis.

Working with a real, large code base, the students observe the benefits of software modelling, including use of modelling tools. They witness the use of what they otherwise may see as "yet another useless thing we are forced to learn" — an unfortunately common student sentiment in Software Engineering courses — as an aid in describing and analysing system architecture.

The deliverable also requires the students to identify (a minimum of three) design patterns used in the system, and to show how each is implemented. They submit a detailed description of each of the design patterns they identified, which must include both structural and behavioural models, and links to the corresponding code.

The study of design patterns in class is timed with the due date for this delivery, so that the students are prepared for the task. Examples of design patterns in the code are also discussed, along with methods of identifying these in a large code base, for example, with the aid of reverse engineering tools. The continual guidance from the teaching assistants additionally ensures that the students know what they are looking for and know how to look for it, when they are working on this part of the deliverable.

Seeing the design patterns they study in class implemented in a widely used software package and witnessing the benefits in a large code base are invaluable sources of motivation for the Software Engineering students.

5.2. Learning to Estimate, Plan, and Manage

For the next deliverable, the student teams are required to select at least two issues (bugs and/or features), complete the implementation of them, and provide suitable test cases to demonstrate that the changes have been correctly implemented. Students have three weeks to work on this deliverable. Most importantly, this part of the project requires students to use their judgement about which change requests to select for implementation. The time available for this part of the project is deliberately tight: the team's goal is to select issues they will be able to solve by the deadline!

The topics covered in class prior to this deliverable include a review of Agile Software Development and

Project Management (these topics are covered in the prerequisite course), Project Planning, and Estimation.

As the first step, the teams examine the list of issues currently open in the project. From the issue list, they select a handful (a minimum of five) of promising bugs or features to examine further. From this shortlist, they need to select a minimum of two items to implement and test. This process involves estimating the effort required to implement each change, and identifying any anticipated risks. The report must contain a detailed explanation of the reasons for selecting the chosen issues.

The weight (marks allocated) for this deliverable is small compared to that of the rest of the project (yet significant enough not to discourage putting in a significant effort). This is an excellent opportunity to make mistakes, underestimate the time required to solve the issues, get ample feedback from the team's teaching assistant, learn from mistakes, and practice Risk Management.

Working with a large, active open source project provides us with an irreplaceable source of excellent exercises on estimation and planning.

5.3. Learning to be a Professional: Soft Skills and Technical Writing

There has been much talk recently about declining levels of maturity and professionalism among university students [28, 29, 30]. If we are failing to teach the students to behave professionally in the classroom, how are we going to prepare them for the workforce [31]?

Many institutions, in particular those awarding an Engineering degree, offer an entire course in professional ethics. In the absence of such a course in the curriculum, educators are left with the task of teaching elements of professionalism in various courses in the program. The course we are describing here proved to be an appropriate home for this topic.

Let us begin with a reminder that the students spend 11 weeks working on a project in a team, where their grade depends both on the team's work and on *their peers' perception of their contribution*. Even if a student does not fully realise the impact of effective communication and collaboration on the final team product, he/she cannot neglect the potentially large adjustment to his/her individual grade.

The team begins by discussing, agreeing upon, writing down, and signing a *Team Agreement* that establishes the ground rules for team collaboration and communication. For example, expected time for an email or message response, dealing with lateness for, or absence from, a team meeting, etc. are part of this agreement. During the term, whenever an issue arises, the students are advised (and helped) to refer back to the agreement. It is critical that help is readily available both from the TA and from the instructor to deal with team conflicts. These very often turn out to be effective learning opportunities to improve the students' soft skills.

Importantly, each contribution may take several days of back-and-forth online discussion with main project developers, and addressing their requests for changes, improvements, etc., until getting a final "Thank You" from the project. We found that the fact that the team needs to effectively communicate with people from "the real world", outside of the university environment, provides a great incentive to think carefully about phrasing questions, explanations, etc. on the project's public forum.

Last but not least, the students need to write! Each deliverable involves a report, which is, in part, marked based on the report's presentation (the report must be well formatted, easy to read, and easy to navigate, with a well-chosen layout), as well as the quality of writing (language, grammar, clarity, and professionalism). The teams have multiple opportunities to improve on their technical writing, following the detailed feedback from the TAs.

5.4. Learning to Verify and Validate

Given that Verification and Validation is a major knowledge area in the ACM and IEEE Computer Society Curriculum Guidelines, and yet is one of the top items in the list of most important knowledge deficiencies of recent graduates [21, 2], it is important for us to provide the students with a solid introduction to the topic.

The majority of the subject matter is covered in the prerequisite course, so the content of the lectures on this topic is largely a review. Luckily, the project naturally provides the students with ample exercise both in automated unit testing and in acceptance testing. The former is required by the open source project: all new code must pass all the tests that are already in place, including checks for correct style of the code, as well as any new ones added to verify the new features or bug fixes introduced by the new code. The latter is required by the project deliverable, as well as, to some extent, by the necessity to document their feature on the project website.

5.5. Synthesising and Applying the Knowledge, and Making a Difference in the "Real World"

In the final project deliverable student teams select, design, implement, test, and document a *new feature* (cannot be a bug fix) for the open source project. This is the most significant — both in terms of workload and in terms of allocated marks — part of the project. The students have five weeks to work on this deliverable.

Each step in this part needs to be explained, documented, and reflected upon. In the selection process, the students practice their estimation skills. In designing the solution, students practice their modelling, design, and evaluation skills: the design of new code must be fully documented and evaluated, and all interaction with existing code needs to be explained. The development must follow the agile development process. The new feature must be fully tested: (a) the open-source project requires every new contribution to be accompanied by new tests that must be added to the existing, extensive, automated test suite, and (b) the students are required to submit a comprehensive suite of acceptance tests for their feature. Finally, the new feature needs to be fully documented: (a) again, the open-source project requires new features to be documented in their "What's New" section, and (b) for the course, students must explain and defend their design decisions. All in all, completing this deliverable incorporates knowledge and skills learned in the entire Software Engineering curriculum of our program.

6. Results and Experiences

After four offerings of the course, we acquired considerable experience in setting up and running the course, and in maintaining a productive working relationship with the lead developers.

We have not conducted any formal studies, and thus we do not have quantitative data on the results of the course. We nevertheless feel it is valuable to share our observations, both from the university education perspective and from the development community perspective.

6.1. Benefits and Challenges for the Students

We have already mentioned that unfortunately, at present our institution is unable to offer / require a capstone project course. The benefits of completing a capstone project are, of course, numerous [7]. This is an opportunity for the students to integrate and synthesise the knowledge learned in multiple courses and to apply it to a real problem — an essential part of preparing students for the workforce.

We believe that the course project we described here is, at the very least, the next best thing to having a separate capstone project course. In fact, it demonstrates all of the accepted benefits of a capstone project:

- It constitutes an authentic, project-based activity that closely relates to professional work in the field.
- Students must apply the discipline knowledge and skills acquired in their program, as well as generic skills.
- It helps students develop communication skills.
- It facilitates students producing "explicit evidence of complex and sophisticated graduate capabilities".
- Last, but not least, it enables the instructor / department to assess the student's final graduate capabilities, as well as to assess and reflect on the effectiveness of our programs.

In addition to the above benefits, we believe that our implementation of the course project serves as a great motivating factor for our students. Research shows (see Section 2) that both Project Based Learning and Service Learning can greatly increase the levels of student motivation. Our personal experience as course instructors strongly suggests that computer science and software engineering students are highly demotivated by working on "fake" projects, "simplified for the classroom" projects, projects that are thrown away at the end of the term.

In our experience, the sole fact that the student code is being reviewed by the lead developers on the open-source project — evaluated by "real" professionals outside of the university setting — encourages a great sense of ownership of and responsibility for the product of students' work. The realisation that people all around the world might, at this very moment, be using the code written by their team, fills the students with pride for their work. And, of course, it provides them with something awesome to put on their resumes and stand out among recent graduates.

As of today, the students from our course have made over 80 pull requests to the FOSS project. To the best of our knowledge (from the instructor's record keeping in each course offering), not a single request resulted in no communication between the lead developers and the students. The amount of the communication and the time span of the communication varied tremendously. From a quick "Thank you" and merge, to requesting

stylistic changes in various pieces of the code, to changes in the way the PR was submitted (for example, merge vs rebase in git), to larger design-level changes. The longest that we have recorded was 2.5 weeks of such back-and-forth. In the early offerings of the course, in several cases the instructor had to submit the students' grades before the PRs were merged. In this case, it was explained to the teams that if they choose to follow up (and they were encouraged to do so), then a grade amendment would be submitted once the contribution is made. In all of these cases, the students were already communicating with the developers and making the required changes, by that time, and every single one of these teams followed up and received amended grades after the term ended. In the more recent offerings, when the instructor and the lead developers established a partnership, and the coordination was greatly improved, all the students' grades were ready on time for submission.

The teams get extensive coaching from their team TAs on this part of the course work, the "failure" rate (i.e., the number of pull requests not accepted by the project is low: the instructor has 9 instances recorded). In 4 cases, these decisions came from disagreement with the design decisions made in these contributions. In particular, the effects that these decisions may have on future development plans (with which the lead developers are familiar, but neither the students nor the TAs / instructor are). One pull request generated a long discussion among the developers who disagreed on the future course. All of these were excellent learning opportunities for the teams, who had first-hand experience witnessing how such decisions are made in a large project, what the arguments are, etc. As for the assessment part, these teams got the marks for the PRs as if they were accepted by the project, and the instructor explained why their work was recognised this way. In 5 cases, it was the team's choice not to follow up and make the required changes. To the best of our understanding, the reasons were either prioritising other courses over working on this course (e.g., finishing a project in a different course, which was due at the end of the term), or what looked like lack of motivation on the part of the students. As unfortunate as we feel it is, we acknowledge that such cases are probably inevitable given the large class sizes and the wide variance in students' motivation. Since getting a PR accepted is not mandatory in our course, and results in a bonus mark, not every student is going to be motivated to put in the work.

As of today, the students from our course made 60 contributions to the code base. At this point we should mention that the FOSS under discussion has very high standards, and every single PR is thoroughly examined by at least one (most often two) lead developers. In addition, the coding style guidelines, as well as guidelines for working with version control are very strict. We believe that the students, having witnessed the level of scrutiny applied to their contributions, have extra reasons to be proud of their work.

We should also note that although most of the contributions were either bug fixes or minor features, several contributions were complex new features, which now appear on the What's New pages of the project's releases. We also note that what we or the project developers deem "a minor feature" is certainly not a minor achievement for the students, for whom the main challenge is learning to work with the real, very large code base.

The remaining difference between submitted and accepted PRs came from having two teams approach the same issue, in different ways. In at least three recorded cases, the lead developers commented on the pros of each approach and encouraged the teams to join their efforts to come up with a joint solution that benefits from each team's ideas. In these cases, the teams and their TAs were instructed to work together to produce a single PR. All of these were accepted into the code base.

Students are encouraged by the developers' comments:

- *This looks like a great PR...*
- *... this feature makes a significant improvement...*
- *... [thumbs up emoji] — this is an excellent piece of work with a test to boot — great stuff!*
- *Thank you for your work! Hopefully we will hear from you again.*
- *... [thumbs up emoji] on tracking this one down, this looks subtle...*

In the end, the students report on having had a positive experience as a result of integrating work on the open-source project into the course delivery:

- *I think it is absolutely great that we got to work with a real open source project like that...*
- *One of my favourite courses. The fact that we made real open source contributions meant our project and effort actually mattered which I didn't feel in any other course.*
- *I enjoyed the course project and feel it was a good way to apply most of the topics learnt.*

- *The lectures were informative and interactive. Project was amazing to work with. More time on feature implementation would be nice.*

6.2. Benefits and Challenges for the Teaching Team

Running this course is demanding on the teaching team. The instructor's work begins long before the course starts, as it is essential that the instructor is familiar with the code base and ready to help the students with technical advice.

The course timeline needs to be carefully designed, as just-in-time teaching calls for close integration between lecture contents and project deliverables, tight deadlines, and short turn-around time between the time students submit their deliverables and the time they get their feedback.

The most important task in organising the course is building and managing a competent and efficient team of teaching assistants. TAs for this course require extensive qualifications. In addition to being knowledgeable in the course material, they must possess a deep technical familiarity with the architecture and code base of the FOSS project. Soft skills, such as communication skills, responsibility, and conflict resolution are invaluable for this job, as the TAs must be able to effectively manage teams of developers, carry out the weekly meetings establishing the atmosphere that enables students to bring up any potential problems or roadblocks in time.

In practice we have been successful in developing excellent TA support by providing just-in-time mentoring. First time TAs are coached by either a head-TA, who has been a part of the teaching team for the course at least once before, or by the course instructor. The coaching involves detailed discussion of each deliverable, the potential places where teams may struggle, advice on how to keep teams focused and on-track, and a live example of how team interviews are carried out so that the TAs can see what they will need to do with their teams.

Detailed rubrics are provided for each deliverable and for each interview. The level of detail goes right down to the list of questions that will be asked during the interview and to the procedure by which each team member will be required to participate at least once. For marking, the head-TA or course instructor provides examples of fully marked deliverables, showing how work of different levels of quality should be evaluated, illustrating the right amount and tone of the feedback given to the students, and providing a baseline to compare work submitted by different teams.

Once the marking is complete, all TAs get together, discuss the marking for each team, identify potential problems that may need attention by the course instructor, and ensure consistency of both marking and feedback across the board.

This involves a lot of work for the members of the teaching team, but it is also incredibly rewarding. The TAs gain invaluable experience in managing teams of developers, identifying and addressing teamwork issues, and motivating students to work more effectively and at a higher level of quality. A good TA can turn a dysfunctional team into a capable unit able to deliver solid work, and the continued and close working interaction between the teams and their TAs builds a sense of comradeship that is not found in other courses in our program. Often the students will freely share with their TAs their thoughts about the course and the project, providing timely feedback on the course delivery.

6.3. Benefits and Challenges on the FOSS Project Side

In this Section we report on the perspective of the collaborators from `matplotlib`, the open-source project used in the course.

The main benefits to `matplotlib` are three-fold. Firstly, the project receives direct contributions, often of great quality, as judged by the lead project developers and reported on the PR pages of the project. Secondly, on-boarding higher education students over generic first-time contributors reduces the burden on the main developers. Lastly, the collaboration cultivates new regular contributors to the project.

The FOSS project described in this work is widely used in data-science. It has a significant installation base across many domains and a large code-base and API surface developed over 15 years, primarily by volunteers.

The project has commensurately large back logs of both bug reports and feature requests. In 2017 the project saw approximately 200 unique contributors, and 160 of those were first time contributors to the project. Many of those were first-time contributors to *any* open source project and required coaching from the project developers on Software Engineering practices. Student teams, who have support "at home" for basic skills, such as version control, testing best practices, and work planning, can have a real impact on improving the project with a reduced burden on the core developers.

Having had the support from the institution, the students will likely have a better first contribution experience than a generic first contributor, thus increasing the odds that they will become a continuing

contributor and member of the community.

The primary resource provided by the project developers to the students is peer-review — one of our scarcest resources. In terms of absolute numbers, the current class size does not provide a problem. However, due to the timeline of the course, the rapid influx of pull requests can be challenging to review in a timely manner.

Since these contributions are part of course work, rather than self or employer driven, if the pull requests are not reviewed and merged in the timeline of the course, it may be less likely that the authors will follow up after the end of the course. It is possible that the scarcity of resources will become more of a problem as the course continues to grow.

Overall, this collaboration is a win-win for both `matplotlib` and the students, and the developers would like to see such collaborations more widely implemented.

7. Conclusion

We reported on the design and delivery of a senior Software Engineering course, within the limits of a Computer Science program. We showed how a collaboration with a large, active Free Open Source Software project structure allows us to

- incorporate principles of Project Based Learning and of Service Learning, reaping the benefits of these pedagogies,
- effectively, in a hands-on approach, teach a number of essential topics in Software Engineering,
- provide the students with a capstone project experience, given the lack of a capstone project course in our curriculum, and
- use the project as a powerful motivating factor for the students.

We describe that, given a well-planned structure of the course, the integration of a Free Open Software Source project into the course work provides invaluable opportunities and positive experiences for all parties involved: for the teaching team, for the students, and for the open-source community.

References

[1] A. Radermacher, G. Walia, and D. Knudson, "Investigating the skill gap between graduating students and industry expectations," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pp. 291–300, 2014.

[2] A. Radermacher, G. Walia, and D. Knudson, "Missed expectations: Where cs students fall short in the software industry," *Software Education Today*, 2015.

[3] L. Briand, "Software documentation: How much is enough," in *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, IEEE, 2003.

[4] S.Huang and S.Tilley, "Towards a documentation maturity model," in *Proceedings of the 21st annual international conference on Documentation*, ACM Press, 2003.

[5] P. C. Blumenfeld, E. Soloway, R. W. Marx, J. S. Krajcik, M. Guzdial, and A. Palincsar, "Motivating project-based learning: Sustaining the doing, supporting the learning," *Educational Psychologist*, vol. 26, no. 3-4, pp. 369–398, 1991.

[6] R. Pucher and M. Lehner, "Project based learning in computer science — a review of more than 500 projects," in *Proceedings of the 2nd International Conference on Education and Educational Psychology*, 2011.

[7] R. F. D. Jr, "A survey of computer science capstone course literature," *Computer Science Education*, vol. 21, no. 3, pp. 201–267, 2011.

[8] D. F. Levia and S. M. Quiring, "Assessment of student learning in a hybrid pbl capstone seminar," *Journal of Geography in Higher Education*, vol. 32, no. 2, 2008.

[9] R. H. Todd and S. P. Magleby, "Elements of a successful capstone course considering the needs of stakeholders," *European Journal of Engineering Education*, vol. 30, no. 2, pp. 203–214, 2005.

[10] G. H. L. Pinto, F. F. Filho, I. Steinmacher, and M. A. Gerosa, "Training software engineers using open-source software: The professors' perspective," in *2017 IEEE 30th Conference on Software Engineering Education and Training*, IEEE, 2017.

[11] J. L. Warren, "Does service-learning increase student learning?: A meta-analysis," *Michigan Journal of Community Service Learning*, pp. 56–61, 2012.

[12] L. Simons and B. Cleary, "The influence of service learning on students' personal and social development," *College Teaching*, vol. 54, no. 4, 2009.

[13] J. M. Conway, E. L. Amel, and D. P. Gerwien, "Teaching and learning in the social context: A meta-analysis of service learning's effects on academic, personal, social, and citizenship outcomes," *Teaching of Psychology*, vol. 36, no. 4, pp. 233–245, 2009.

[14] J. Eyler, "Reflection: Linking service and learning — linking students and communities," *Journal of Social Issues*, vol. 58, no. 2, pp. 517–534, 2002.

[15] K. Lambright and E. Y. Lu, "What impacts the learning in service learning? an examination of project structure and student characteristics," *Journal of Public Affairs Education*, vol. 14, no. 4, pp. 425–444, 2009.

[16] J. A. Hatcher, R. G. Bringle, and R. Muthiah, "Designing effective reflection: What matters to service-learning?," *Michigan Journal of Community Service Learning*, vol. 11, pp. 38–46, 2004.

[17] L. H. Jamieson, "Service learning in computer science and engineering," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, (New York, NY, USA), pp. 133–134, ACM, 2002.

[18] L. D. Webster and E. J. Mirielli, "Student reflections on an academic service learning experience in a computer science classroom," in *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education*, (New York, NY, USA), pp. 207–212, ACM, 2007.

[19] T. Dahlberg, T. Barnes, K. Buch, and K. Bean, "Applying service learning to computer science: attracting and engaging under-represented students," *Computer Science Education*, vol. 20, no. 3, pp. 169–180, 2010.

[20] J. T. F. on Computing Curricula IEEE Computer Society Association for Computing Machinery, "Curriculum guidelines for undergraduate degree programs in software engineering," 2014.

[21] A. Radermacher and G. Walia, "Gaps between industry expectations and the abilities of graduates," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pp. 525–530, 2013.

[22] J. M. Twenge and K. Donnelly, "Generational differences in american students' reasons for going to college, 1971–2014: The rise of extrinsic motives," *The Journal of Social Psychology*, vol. 156, pp. 620–629, 2016.

[23] G. Novak, E. Patterson, A. Gavrin, and W. Christian, *Just-in-Time Teaching: Blending active Learning and Web Technology*. Saddle River, NJ: Prentice Hall, 1999.

[24] A. Tafliovich, J. Campbell, D. Zingaro, F. Estrada, and L. Porter, "Forming strong and effective student teams.," in *Proceedings of the 48th ACM Technical Symposium on Computer Science Education*, (New York, NY, USA), ACM, 2017.

[25] R. Lingard and E. Berd, "Teaching teamwork skills in software engineering based on an understanding of factors affecting group performance," *Frontiers in Education*, vol. 3, 2002.

[26] A. Tafliovich, A. Petersen, and J. Campbell, "Evaluating student teams: Do educators know what students think?," in *Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, (New York, NY, USA), pp. 181–186, ACM, 2016.

[27] A. Tafliovich, A. Petersen, and J. Campbell, "On the evaluation of student team software development projects," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, (New York, NY, USA), pp. 494–499, ACM, 2015.

[28] P. Gray, "Declining student resilience: A serious problem for colleges," 2015.

[29] N. Howe and W. Strauss, *Millennials Go To College: Strategies for A New Generation on Campus*. Lifecourse Associates, 2007.

[30] A. F. Keaton, "Teaching students the importance of professionalism," in *The Teaching Professor*, (Atlanta, GA), Magna Publications, 2015.

[31] K. L. Campana and J. J. Peterson, "Do bosses give extra credit? using the classroom to model real-world work experiences.," *College Teaching*, vol. 61, no. 2, pp. 60–66, 2013.