Learning professional software development skills by contributing to Open Source projects

Igor dos Santos Montagner
Computer Science and Engineering staff
Insper
São Paulo, Brazil
igorsm1@insper.edu.br

Andrew Toshiaki Nakayama Kurauchi
Computer Science and Engineering staff
Insper
São Paulo, Brazil
andrewTNK@insper.edu.br

***Abstract*—This Innovative Practice Full Paper describes Open Development, a Software Engineering advanced course that focuses on building practical development skills by contributing to Free/Libre and Open Source Software (FLOSS). Software Engineering is a broad area that spans many different topics, such as testing, software quality and development methodologies. Typically, entire courses are developed to explore each topic in depth. However, effective participation in real software projects requires the simultaneous application of a wide range of skills and focusing exclusively on each skill one at a time may not be sufficient for students to be able to apply them all in sync. Participation in FLOSS projects offers a unique opportunity to obtain and develop such skills, as many projects follow industry best practices for all contributions and provide documentation to help new contributors.**

We present the Open Development's design in alignment to Student Centered Learning and the GAPA (Goals, Activities, Projects, Assessment) framework. We also analyze student contributions from the last three offerings (N=50), studying characteristics such as type of contribution (test, bug fix or new feature), complexity of the code, approval ratio and project size. The design and outcomes of the course are validated by examining Student Evaluations conducted by the institution's Teaching and Learning Office. Students from all offerings (N=34) strongly agree that "This course's contents will help me in a future job or internship". Combined with accepted contributions to well-known projects such as Pandas, Matplotlib and Pygame, we conclude that Open Development fulfils the objective of providing practical software development skills.

***Index Terms*—Software Engineering Education; Open Source Software; Professional Skills**

I. Introduction

The Software Engineering industry is very dynamic, with new technologies, processes and best practices being developed constantly. It is also a very broad academic field for both research and teaching. An approach frequently adopted for teaching is the creation of courses that focus on a single aspect of the field, such as testing [1], [2], Functional Programming [3], Agile [4] or Database Systems [5]. This type of focused course has the advantage of isolating concepts, which makes it easier for students to learn these subjects in their first contact. However, there are also limitations.

The first one is that in practice the boundaries between topics are not clearly defined. A task in Software Development might include implementing a new feature, database migrations, writing unit tests and documentation and requesting code reviews from more experienced developers. A single focused course is not likely to provide an environment to develop this kind of skill, since they are focusing on a single part of the process. Thus, it is left for the student to join the pieces at a later moment, possibly by themselves.

A second limitation is that instructors oftentimes are not active Software Engineers themselves, so they might not be able to offer quality guided instruction in current and modern tools, frameworks and processes. Assessment of the technical side of projects may be a problem as well, since the instructors might not be up to date with the current best practices of the industry.

The *Open Development* course was created to address these limitations and act as a complementary course to the single focused ones. Its objective is to develop practical Software Engineering skills by contributing to Free/Libre Open Source Software (FLOSS). This provides unique opportunities to develop practical skills, as many follow best practices for code quality, testing, documentation and code reviews. This also includes working with Version Control Systems (VCS) in a remote and distributed manner and receiving feedback from more experienced developers and maintainers. Students are free to choose projects of their liking and are encouraged to communicate with maintainers and other contributors as much as they can.

The course is split into three major parts: tutorial, supervised contributions and project. In the first part students work in groups of at most three students and learn introductory material on a diverse set of preparatory subjects, such as unit testing, documentation tools and VCS. The objective is not to dive deep into each one, but only teach enough so students can progress by themselves when the need arises. In the second part students continue working in groups, but now are required to choose a project and send a code contribution. Many suggestions are presented by the instructors, but students are free to choose any project they like. Students are not expected to work on every technical detail by themselves and are encouraged to ask for help from their peers and instructors. The final part is a project-based experience where students choose which contributions they would like to make from a large list of possible (and optional) tasks. To submit tasks for consideration students must send a well-formed Pull Request

(PR) to the course's repository. Thus, from day one students are practicing how to collaborate and communicate using VCS and related tools.

Results from three offerings of the course ($N = 50$) show that (i) students are able to contribute to well-known projects such as Pandas, Pygame and ReadTheDocs; and (ii) students recognize that the course's contents are aligned with the skills they will need in a future job. We also analyze data from the student contributions, focusing on how choice seems to affect students and their engagement in the course.

In section II review in details relevant previous works in Software Engineering Education that include contributions to FLOSS projects. In section III we give an overview of the design of the course, including Learning Objectives, Activities and Assessment. In Section IV we describe in detail how the student proficiency is assessed using a gamified badges-like system. In section V we analyze student perceptions about the course and their outcomes. Finally, in section VI we present our conclusions and future work.

II. FLOSS in Software Engineering Education

Previous works have used FLOSS in Software Engineering Education in many different forms [6], [7]. In this section we review works that are particularly important given the context of *Open Development*.

Spinellis [8] argues that contributing to open source projects allows students to acquire professional skills that are hard to emulate in traditional programming assignments, such as interacting with a global and diverse community and addressing code review comments. Students can also train a multitude of technical skills that are not usually part of traditional Software Engineering courses, from configuring build environments to creating documentation using markup and following sophisticated version control workflows including branching and rebasing.

Marmorstein [9] describes a project component in a Software Engineering class. Students have freedom to choose projects and issues to work on, being awarded different amounts of points for features, bug fixing/reporting and other types of contributions. The project also requires students to design a work plan for the semester and to research important information about the projects and communities they are trying to interact with. Surveys with students reveal successful experiences in working with professional tools, communication and the importance of documentation.

Diniz et. al. [10] proposes the use of a gamified environment to help students overcome orientation barriers related to the process of contributing to open source software. Four game elements were introduced: Quests, Points, Levels and Ranking. Quests represent a step-by-step process that students must complete in order to send their contributions. Points are used to provide feedback to students that a task was completed. Levels were used to orient students to focus on a given step before proceeding to other challenges. Some Quests were only available at certain Levels, so students would not be able to skip steps. A Ranking was created to motivate students through earning prestige with their peers. Experiments with 17 students working on a single project (JabRef) were encouraging. Quests and Points were seen by students as the most helpful elements both for helping students orient themselves in a new project and as motivation. An important issue in this work is that students contributed using a modified Gitlab environment separated from the "official" repository and that contained their customized gamification elements.

Krutz et. al. [2] describes a Software Testing course in which students use software testing tools, techniques and processes to test a free software project of their choice. Teams of up to 6 students produce a detailed report containing all issues found using several different types of testing, such as Unit, Acceptance, Usability and Compatibility. Although not mandatory, most teams decided to share their findings with the software project they chose. It was not expected for students to propose fixes for the issues found. Students were encouraged to provide step-by-step instructions to reproduce the issues.

Open Development shares many similarities with these previous works, but also diverges in important directions. First, differently from [10] students are free to choose projects to work on and use a standard Github environment. This is an important limitation of [10]: one of the authors is the maintainer of the chosen project, so students never get to interact with a diverse community, only between themselves and one of the authors of the study using a separated customized Gitlab instance. In terms of game elements, we maintain the use of Points and replace Quests with *Skills*. Unfortunately, Diniz et. al. does not include a description of the Quests proposed, so we cannot compare them to our use of *Skills*. From the only example of Quest provided we infer that Quests are procedural, more akin to a list of tasks than to the description of an ability. Earlier versions of the course used Ranking and Levels, but these were dropped when students reported that they were not engaging.

We also emphasize repetition more than all three analyzed works. In the case of [8], contributing to open source is an assignment in a larger "Software Engineering in Practice" course and in [10] the study lasted only 20 days. To be proficient in the process of contributing to a project students need to repeat it many times and both [10] and [8] imply that only a single contribution was expected. Since we expect more repetition and time is finite, this also means that sometimes student contributions in *Open Development* are not especially complex additions. Spinellis shares some similar experience: "To an undergraduate student, the barriers to open source contributions are often so high, that getting 20 lines of code integrated into a large project is a worthwhile achievement indeed" [8]

In terms of expected learning outcomes, Spinellis presents a long list of possible technical and professional skills that can be obtained by contributing, but it is not clear how these are assessed. Diniz et. al. does not describe what the expected student performance was, only that students were not expected to have code merged into JabRef. It's not clear how close to getting code accepted students were nor which tasks students

were expected to be proficient in. Spinellis is clearer in this regard: getting contributions merged is not expected but graded positively. Marmorstein describes learning outcomes clearly but did not directly relate them with the assessment of the project. Student surveys were used to evaluate if the learning outcomes were achieved. Krutz et. al. [2] has clear student outcomes but are focused on Software Testing skills only. This is not a problem per se, given that the course project is done in the context of Software Testing and fixing code is not part of its learning objectives. However, students are not able to experience the entire process from finding a bug to fixing it.

The differences outlined above set *Open Development* apart from these earlier experiences. We present student centered learning objectives and a rubric that maps student performance to a proficiency level. Requirements for each level are clear and describe what a student at that level can do. Repetition is a requirement for more advanced levels of mastery and is an important part of learning. We also do not restrict ourselves to a single type of contribution (such as testing tools or code contributions) nor to a single project: students have freedom to choose both projects, the type of contributions they would like to do and are encouraged to reach for people in the community for help and mentoring.

III. The design of Open Development

The course's design is supported by the following principles.

- *Active Learning*: although some sessions include a small lecture at the beginning, in most sessions students work on the proposed tasks with the support of the instructor and/or their peers.
- *Controlled choice*: students are free to choose projects to contribute to. A large number of optional tasks is provided and students choose how many and which ones they work on. A small number of mandatory tasks is present to ensure that students have achieved the Learning Objectives.
- *Grading transparency*: Every student outcome (activities and project outcomes) contains a description of what **Done** means. Feedback is given both by the instructor and maintainers of FLOSS projects. Students know their final course grade at every point in the semester.

Open Development was designed using *Backward Design* principles proposed by Wiggins et. al. [11]. This framework states that we, as course designers, must first identify which are our desired results. Then we determine how we measure if a student has achieved these results. Finally, we plan learning experiences that can prepare students to perform at the level we determined in the previous step. To ensure that the course remains consistent with the design we employ the GAPA framework [12] proposed by Stolk and Martello. GAPA proposes the integration, consistency and constant revision of a course's **G**oals, **A**ctivities, **P**rojects and **A**ssessment.

The goals of the course are stated below as Student Centered Learning Objectives ([13], Appendix D). Each Learning Objective (*LO*) describes what a student should be able to do by the end of the course. In our case, a student that is approved in *Open Development* should be able to:

- **LO I** - analyze an unknown medium to large code base and modify it to fix bugs or implement new features;
- **LO II** - interact with a remote development team to deliver code that fulfils minimum software quality and style requirements;
- **LO III** - understand software distribution and licensing

Assessment is done using a gamified system where students do various tasks related to the course's _LO_s to earn *Skills* and *Experience Points (XP)*. A grading rubric (Table V)) is used to indicate how much *XP* and which *Skills* are mandatory for each proficiency level. More details are presented in Section IV.

Open Development starts with a set of Preparatory Activities that provide a first look at a topic, such as unit testing, documentation or guided bug fixing. They are mostly done in groups of at most three students with a deadline around two thirds of the semester. All *Skills* in the *Tutorial* category are Preparatory Activities. These start with a discussion about the topic to understand students' prior knowledge on the topic and finish with a presentation or toy project that incorporates basic aspects about the topic.

The course's Project is centered around contributions to FLOSS projects, including bug fixing, creating new features and improving documentation and translation. Since difficulty varies in these tasks, students can repeat most *Skills* and repeating the same *Skill* multiple times yields more *XP* per execution.

The course's schedule consists of three phases, *Tutorial*, *Guided Contribution* and *Project*. These are detailed in the following sections.

A. Tutorial

During this phase students complete introductory tasks in the following topics:

- Version Control Systems: forking repositories and creating Pull Requests; advanced git operations (*rebase*, *squash* and *commit editing*);
- Software quality: software testing, linting, documentation;
- Distribution: packaging python libraries, Docker;
- Software communities: communication channels, leadership, funding, governance.

All these activities are mandatory for passing the course and can be done in groups of three students. After completing these tasks students should be familiar with most tools used in large-scale software projects.

B. Guided Contribution

In the *Guided Contribution* part of the course students create their first code contributions. Steinmacher et. al. [14] identified 15 barriers faced by newcomers to FLOSS projects. Since most students are newcomers to FLOSS projects, helping students find their way is the main objective of this part. Many of

these barriers can be removed by having the instructors actively guide students during class. This includes the following measures.

- groups of at most 3 students can work on a single problem. This helps students that tend to get stuck finding an issue to work on or do not have great debugging skills.
- instructors create a short list of issues and projects that are known for being receptive to newcomers
- students receive feedback on which issues they want to work on and are guided to avoid some of the barriers listed
- activities during class that specifically treat some of the barriers listed

The barriers to first contributions listed in [14] are grouped into five categories: Technical Hurdles, Documentation, Social Interaction, Previous Knowledge and Finding a Way to Start.

Having observed many barriers during our experience in the course, we developed activities to discuss how to overcome the five groups of barriers identified in [14]:

1) Social Interaction: students choose a project and research information about the community of people around a project. Students must answer questions like "Where is the code hosted?", "How to determine if a project is active?" or "Where do I get help if needed?", "How is the development team organized?" and "Who are the core team members?". Then students give 5 minute presentations summarizing their research. Projects are chosen by the instructors to ensure diversity.
2) Previous Knowledge: students receive software without documentation and must read code and understand how to run it. Students follow a worksheet that guides them into discovering what they would need to study in order to run (and contribute to) and project. The emphasis here is on the process of reading code and identifying the missing pieces of knowledge.
3) Technical Hurdles and Documentation: the instructor selects an issue/project and proceeds to create a development environment in a dialogic teaching manner. Every time an error or problem occurs the instructor proceeds to study the documentation with students, showing the process of how to obtain information from it.
4) Finding a way to start: the instructor explores a codebase looking for clues on how to reproduce an existing issue and where the problem might be occurring. This usually involves collaborative debugging using visual tools to get insights on how the code is running before it fails. This is a dialogic activity that alternates between the instructor modelling how to explore a codebase and students giving input on the next steps of the process and where to look for more information.

During the *Guided Contribution* part at least half of the classroom time is dedicated to supporting students in their contributions.

C. Project

During the final month of the course students start the *Project* part where they have full autonomy to choose projects, issues and *Skills* they want to earn. By this time students are already aware of most barriers to contributions and how and where to ask for help.

All classes in this part are dedicated to supporting students in their work. Different from what one might expect, helping with code is not the most important requests students make. Although code issues are frequent, pair debugging with instructors usually helps. Students are already familiar with the tasks at hand (fixing a bug or implementing a new feature) but have only worked on much smaller codebases up to this point. The two most critical issues raised by students are related to the barriers "Finding where to start" and "Social interaction"

Many students struggle to define a project to contribute to. Not only do most mature projects have hundreds of open issues, but most are also not fit for newcomers. Recently many projects have started labelling issues as *Newcomers*, *Good first issue* or *Help wanted* in order to direct newcomers to more approachable tasks for their first contributions. This usually involves the decision of an experienced member of the project to actively label issues.

Students also struggle when interacting with project members. Many students report sending messages trying to discuss features or ideas and not receiving answers, while others are bothered by the amount of time project members take to respond to PRs. A significant portion is also afraid of saying something "stupid" without realizing.

Tackling these barriers with students usually involves setting reasonable expectations for communication and limiting the choice to projects that were already used in other course offerings. This is currently done by talking with students one at a time and listening to what they have tried and what they would like to achieve.

IV. Assessment using Skills and XP

In *Open Development* assessment is done based on *Skills* that students obtain by doing coursework. Completing a *Skill* awards students a certain number of *Experience Points* (*XP*). Every *Skill* is linked to one of the _LO_s described in Section III and represent evidence of mastery in it.

All course materials are hosted on Github. Each student has an individual cryptographed file that contains all their submitted coursework. To complete a *Skill* students must edit their achievements file and send a PR to the course's Github repository. The instructor then checks the result and either (i) asks for modifications; or (ii) merges the PR. If students do not reply within two weeks, the PR is closed and they must start the process again. Discussions are held using PR comments. This process for submitting coursework mimics the *Fork and Pull* workflow used by many large software projects. Therefore, even before they start working on external software projects students are already learning to collaborate using VCS.

Proficiency	Description	XP
First Steps	Sent first PR to the course's repo	2
Exploring a project	Researched the community of people and services around a software project	3
Uncharted project	Run a simple software project without instructions	5
Professional Project	Refactor bad code and create project landing page	5
Tested and Approved	Created unit tests and Selenium tests for a web project	5
Python Package	Created a python package installable via pip	3
Dockerfile	Created a Dockerfile for a webapp written in Python	3
Basic translation	Used translation tools to internationalize a CLI program	2
First code contribution	Sent the first PR fixing an easy bug	5
FLOSS research	Seminar involving recent themes in FLOSS	10

TABLE I
TUTORIAL SKILLS.

Proficiency	Description	XP
Code contribution	PR sent to a project	7
Accepted contribution	PR accepted	13
Simple contribution	A PR with minor changes was sent and accepted	3

TABLE II
CODING SKILLS.

Proficiency	Description	XP
Bug report	Reported a bug and another user was able to reproduce it	5
Feature request	Requested a feature and was acknowledged by a maintainer	2
Indirect issue	Discussions in a PR you sent where moved to a separate issue	3
Beta tester	You tried to reproduce a bug and discovered it was already fixed, causing the issue to be closed.	3
Accepted Translation	A set of translations was accepted	5
Reviewer	Review typos and other small problems in a project documentation	2
Docs contribution	Had a PR accepted in a issue labeled *Docs*	7

TABLE III
COMMUNITY SKILLS.

Proficiency	Description	XP
Speaker	Presented a talk about FLOSS in an event	20
Helpdesk	Answer accepted in StackOverflow	4
Release	One of your contributions is present in a recent software release	10
3 PRs sent	Sent 3 PRs to the same project	10
3 PRs merged	3 PRs to the same project were accepted	30
Self-Promotion	Something that you did was cited or acknowledged by the project in social media	10

TABLE IV
IMPACT SKILLS.

Skills are divided into four categories: *Tutorial*, *Coding*, *Community* and *Impact*. *Tutorial* skills (Table I are the preparatory activities described in Section III and are mandatory for all students. *Coding* skills (Table II) are code contributions to software projects. *Community* skills (Table III) are every other type of contribution related to the community of people (users, developers, activists) around software projects. *Impact* skills (Table IV) are obtained when students' contributions are large enough so other people in the community notices them. This includes receiving public acknowledgement from the community, repeatedly contributing to the same project or having a PR published in a new release. All *Skills* not in *Tutorial* can be done more than once. By design, it is more advantageous in terms of *XP* to repeat some *Skills* such as *Code Contribution* than to try and do every *Skill* once. Most of the *Impact Skills* try to recognize this by awarding points that rely on a more long-term relationship with a single project.

The grading rubric used in the course is presented in Table V. For *Basic* and *Proficient* levels the amount of *XP* required is roughly proportional to the effort required. A student with *Basic* proficiency can contribute to a project both with code and docs or translations. Since the first contribution is part of the Tutorial *Skills*, in *Basic* a student is involved in at least 2 PRs.

Notice that only sending a PR is required. This is done for two reasons: i) some projects take a long time to actually accept a PR and this time frame might not match with the course's; and ii) frequently maintainers point places where the PR must be improved before merging and fixing these takes more effort than the initial PR. This additional step of making a PR good enough for merging is required for *Proficient* students and represents adequate mastery of the _LO_s.

The *Advanced* level requires so much more *XP* that the only way of achieving it is to obtain *Impact* skills. This is consistent to the course's _LO_s: an advanced student should be able to repeatedly contribute to a project to a point where

Proficiency	XP	Mandatory Skills
In Development	0	Complete all *Tutorial* tasks
Basic	50	1 *Code Contribution + Docs contribution* **OR** *Software Translation*
Proficient	90	1 *Accepted Code Contribution*
Advanced	150	

TABLE V
GRADING RUBRIC BASED ON TOTAL XP AND EXTRA REQUIREMENTS.

their contributions start being noticed by the community.

V. STUDENT OUTCOMES AND PERCEPTIONS

In this Section we discuss students' perceptions and outcomes from *Open Development*.

A. Student perception of the course

At the start of the semester students are polled about their motivations, desires and perception of the course. The following questions are asked:

1) **Why did you join the course? (Multiple choice)** Half of students report that "Learn to contribute to open source software" is their main motivation. "Learning a new technology", "Recommendation" and "Convenient schedule" shared the remaining answers equally.
2) **Cite some technologies you would like to learn (Free response)**: The majority of students (about 60%) would like to use Python in the course, with Rust coming in second place. A word cloud of all answers is shown in Figure 1.
3) **Which platforms would like to contribute to? (Multiple choice)**: Students prefer to work on libraries (83%), followed by Web and Desktop Linux (about 60% each). Other possible answers were Desktop Windows, Android and iOS.
4) **Cite some projects you would like to work on in this semester (Free response)**: about 40% of students didn't cite any specific project. Students were not expected to follow this throughout the semester and indeed most did not contribute to any project on this list.
5) **What would be level 10 achievement for you in this course? (Free response)**: almost all students cite having code accepted in a project they like/use. Interestingly, having a PR accepted is only at the *Proficient* level and many students either achieve this level or easily surpass it.

At the end of the semester an institution-wide course evaluation is done. Amongst the many questions, two related to professional practice are relevant:

- **The contents of the course will help me in a future internship or job**: evaluated from 0 (completely disagree) to 10 (completely agree), the course's average for all offerings is 8.7.

Fig. 1. Technologies students would like to learn

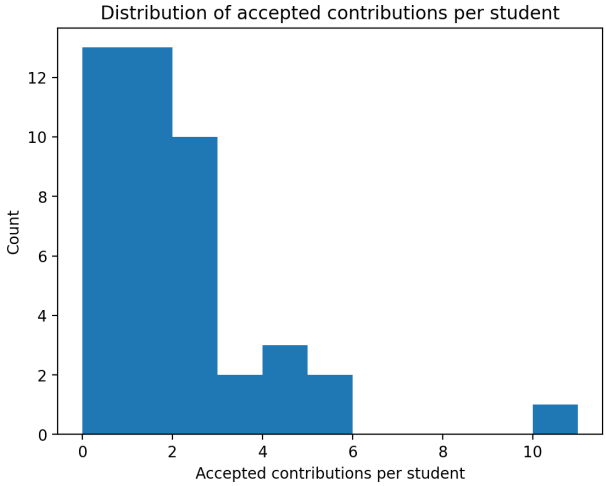

Fig. 2. Histogram of accepted PRs per student.

- **The instructor discusses the relevance of the material (why learning it is important and the connection to practice)**: evaluated from 1 (completely disagree) to 5 (completely agree), the average of all offerings is 4.91.

These indicate that students recognize that the course develops professional skills that will be used in their future software development careers. Students also understand the connection between the coursework and practical skills and applications. From the students' point of view, *Open Development* is a success.

B. Student engagement with FLOSS projects

We collected data from all contributions students sent to projects in the last 3 offerings.

Students sent a total of 122 code contributions in the form of Pull Requests, 72 of which were accepted. Although the acceptance ratio (59%) is consistent with results in [10], students in *Open Development* can submit more than one PR during the course. Figure 2 shows the distribution of PRs accepted per student. Only 13 out of 50 students didn't have any PR accepted in the course, so 74% of enrolled students eventually had at least one PR merged.

The choice of programming language was consistent with students' initial desires: most sent PRs using Python as the main language. The second most used language was Typescript, followed by Java and Rust. A word cloud of all programming languages used is shown in Figure 3. Only Python and Typescript were consistently used by a large number of

Fig. 3. Word cloud of programming languages used.

students. All other languages were used by only a few students and they appear large in the word cloud due to these students sending many PRs.

The project of choice was Pandas, with 27 PRs sent. Pygame was also selected by 5 different students for contributions. All other projects received PRs from at most 2 students, with a total of 56 projects receiving a PR. Pandas was also the most receptive project to newcomers: project members always reply to all PRs with constructive feedback and in a timely manner.

VI. Conclusion and Future Work

Open Development was created to promote training on practical software development skills through contributions to open source software. We have stated the course's learning objectives and detailed the design of the course, including rubrics and a list of all possible tasks students can complete. The course uses game elements to support students' autonomy and provide guidance on which tasks are more valuable as evidence of mastery in the learning objectives. Both student perception and an analysis of the PRs sent by students reveal that *Open Development* achieves its objectives.

We identify several possibilities for future work. Currently some students struggle with the amount of autonomy required in the Project portion of the course. Finding a balance between the structure described in reviewed works [2], [9] and letting students set their own pace is a challenge. We would also like to improve current introductory activities in the Tutorial part, possibly introducing peer assessment (like [10]) and using more contextualized tasks. There is also a problem of scale: all offerings had around 20 students each, but the course's design is not currently scalable for much larger classes without adding more instructors. This is also cited in the reviewed literature as a relevant issue.

References

[1] M. Aniche, F. Hermans, and A. Van Deursen, "Pragmatic software testing education," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 414–420.

[2] D. E. Krutz, S. A. Malachowsky, and T. Reichlmayr, "Using a real world project in a software testing course," in *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014, pp. 49–54.

[3] A. Hameer and B. Pientka, "Teaching the art of functional programming using automated grading (experience report)," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–15, 2019.

[4] J. Campbell, S. Kurkovsky, C. W. Liew, and A. Tafliovich, "Scrum and agile methods in software engineering courses," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 319–320.

[5] E. D. Canedo, I. N. Bandeira, and P. H. T. Costa, "Challenges of database systems teaching amidst the covid-19 pandemic," in *2021 IEEE Frontiers in Education Conference (FIE)*, 2021, pp. 1–9.

[6] J. Nandigam, V. N. Gudivada, and A. Hamou-Lhadj, "Learning software engineering principles using open source software," in *2008 38th Annual Frontiers in Education Conference*, 2008, pp. S3H–18–S3H–23.

[7] D. M. Nascimento, K. Cox, T. Almeida, W. Sampaio, R. A. Bittencourt, R. Souza, and C. Chavez, "Using open source projects in software engineering education: A systematic mapping study," in *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2013, pp. 1837–1843.

[8] D. Spinellis, "Why computing students should contribute to open source software projects," *Communications of the ACM*, vol. 64, no. 7, pp. 36–38, 2021.

[9] R. Marmorstein, "Open source contribution as an effective software engineering class project," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 268–272.

[10] G. C. Diniz, M. A. G. Silva, M. A. Gerosa, and I. Steinmacher, "Using gamification to orient and motivate students to contribute to oss projects," in *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2017, pp. 36–42.

[11] G. Wiggins, G. P. Wiggins, and J. McTighe, *Understanding by design*. Ascd, 2005.

[12] J. Stolk and R. Martello, "Reimagining and empowering the design of projects: A project-based learning goals framework," 10 2018, pp. 1–9.

[13] S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman, *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons, 2010.

[14] I. Steinmacher, M. A. G. Silva, M. A. Gerosa, and D. F. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," *Information and Software Technology*, vol. 59, pp. 67–85, 2015.