# Teaching Undergraduate Software Engineering Using Legacy Code

Phill Conrad
phtcon@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Andrew Lu
alu@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

## ABSTRACT

This experience report describes the design and implementation of an undergraduate software engineering course centered around working with legacy code. Students in this course contribute to a code base that has been handed down from a previous offering of the same course, and is then passed down to the students in the next offering of the course. We review the literature describing the gap between students' preparation coming out of undergraduate computing programs, and the skills they are expected to demonstrate in entry-level software development jobs. We then describe how our course seeks to bridge each of those gaps. Finally, we describe the challenges of delivering a course centered around legacy code projects, and reflect on our efforts to meet those challenges.

## CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**.

## KEYWORDS

software engineering education, legacy code

**ACM Reference Format:**
Phill Conrad and Andrew Lu. 2023. Teaching Undergraduate Software Engineering Using Legacy Code. [redacted], 7 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Ever since Begel and Simon reported on the struggles of novice developers at Microsoft [2, 3] in 2008, many authors have followed up to take note of the ways in which recent computing graduates often struggle in their first software development job. These studies are often accompanied by suggestions of ways in which software engineering courses may be modified to help bridge this gap.

In this experience report, we describe our efforts to implement many of these suggestions in the context of a 10-week undergraduate software engineering course with enrollments of around 72 students per term. Our course design is centered around a unifying theme: providing students with the opportunity to work with *legacy code*. When the course is in steady state, each time the course is offered, the focus of the course is preparing students to make contributions to a base of code that has been handed down to them by the students in a prior instance of the same course.

This design flows directly from one of the key recommendations of Begel and Simon's work, namely that instructors provide students "a large pre-existing code base to which they must fix bugs (injected or real) and write additional features". In this paper, we survey the related work (starting with Begel and Simon) to characterize the gap between student preparation and industry needs. We then describe our design, and ways in which it addresses this gap. We then reflect on the challenges of delivering a course in this format.

## 2 RELATED WORK

We highlight three areas of related work: (1) work that examines the gap between students' preparation and the needs of industry, (2) work that looks at incorporating legacy code into software engineering courses, and (3) work that explores design issues in project-based courses.

### 2.1 The Skills Gap

Previous work has explored the the gap between student's preparation and the tasks they are expected to perform in their first job—and this work has often suggested the very approaches that we are taking in the course described in this experience report.

In the mid 2000s, Begel and Simon [2, 3] studied eight novice developers during their first six months at Microsoft and concluded that while "university computer science curricula provide [novice software developers] with adequate design and development skills, their communication, collaboration, and orientation skills are not as well addressed." They also concluded that "many of the problems they have typically have a root cause in poor communication skills and social naïveté."

In 2013 Rademacher and Walia [25] conducted a structured literature review examining the areas where "graduating students ... [fell] short of the expectations of industry or academia". Like Begel and Simon, they found gaps in students preparation in soft skills, but they also uncovered gaps in technical preparation as well, including design, testing, and configuration management tools.

Exter [10] performed a 2014 study of designers/developers of educational software, and found gaps in "testing, maintaining code over time, use of source code control and development tools... communication, critical thinking and problem solving". The participants in this study suggested that it would help if instructors could "bring existing large-scale real-world applications and infrastructures into the curriculum". Exter also noted this (emphasis added):

> One of the more unique findings in this study was the stress participants placed on real-world experiences of significant duration beginning early in the program and continuing throughout it (larger than a typical one or two semester capstone). These authentic, large-scale project based experiences would ideally offer students the chance to engage with tools and techniques used in practice, interact with cross-disciplinary teams, and develop critical thinking and problem solving skills. *Significant structural changes would be necessary to allow for this type of experience within most undergraduate programs.*

Exter goes on to describe some of the structural changes undertaken at various institutions: (1) a four-semester long "Software Studio" sequence [23], (2) a service-learning program that students can participate in for up to eight semesters [8], (3) a four-year curriculum integrated with work experiences [7]. Later efforts include work by Carmichael et al. on four-year curriculum that is integrated with paid work in industry [5]. Such structure change may be infeasible for many programs; we show a way these types of experiences can be incorporated, at least to some extent, *without* significant structural changes to the academic program.

In 2018, Craig et al. [9] reported the results of semi-structured interviews with recent college graduates, and structured their findings around six themes—we use these six themes in Section 4.

More recently, between 2019 and 2021, Groeneveld et al. performed a structured literature review [15], a survey of course syllabi looking to see which soft skills were taught [13], a Delphi study to determine what soft skills are valued by experts in Software Engineering [14], and finally an analysis of the three prior studies plus eight follow up interviews to really zero in on the best characterization of the gap [16]. They identified the three biggest gaps as: "devoting oneself to continuous learning, being creative by approaching a problem from different angles, and thinking in a solution-oriented way by favoring outcome over ego."

Many other recent papers have echoed this theme of the gap between students preparation and employers needs [20, 28, 29], emphasizing that this is far from a solved problem.

## 2.2 Using Legacy Code in Courses

The Humanitarian Free and Open Source Software Project [4, 21] is a framework in which students work on real-world projects that address humanitarian needs, where the code base may span multiple course offerings. This framework offers many of the same advantages as the course design we present in this experience report—two aspects in particular are the use of a real code base that students contribute to, and an emphasis on authenticity. Indeed, Nascimento et al. [22] specifically looked at whether HFOSS projects help to bridge the preparation gap for industry, and found that they did.

Another open source project approach, UCOSP [17] matches students with open source projects and mentors that are professional developers. Three pillars of the program are that students work on real projects, with real users, and are guided by real (professional) mentors. A study of student surveys from this program shows that among the elements students value from this approach is the exposure to complex systems, non-greenfield code (legacy code), real development processes, code review, reasoning about "real, novel requirements".

One disadvantage of both the UCOSP and HFOSS approaches is the need to build and maintain relationships with external stakeholders (i.e., the consumers of HFOSS software, the professional developers that maintain the software repositories using in UCOSP). In addition to being time-consuming, this places additional external constraints on instructors. Our approach tries to achieve many of the benefits of these approaches while avoiding some of the complexities of dealing with external entities.

## 2.3 Design Issues in Project-Based Courses

Richards [26] presents a literature survey of ten design questions regarding project-based courses, and suggests that the answer to all but one (optimal group size) is "it depends"[1]—as with many complex designs questions, there are tradeoffs rather than clear prescriptions. Assessment of software projects is an issue that is particularly difficult, and about which much has been written, particularly in the context of capstone courses [11, 12, 18, 30, 31]. As we discuss in Section 5, while we have settled on a particular approach to assessment, we recognize that it has strengths and weaknesses.

# 3 DESIGN OF OUR COURSE

There are four main principles that guide the design of the course:

- **Authenticity**: Where possible, we make the software development experience as authentic as possible. This includes mirroring industry practice as closely as possible, and building software that will actually be used.
- **Minimizing Cost**: When using professional tools, we try to limit ourselves to the "free tier" of those tools and/or to tiers that are free to verified educational users.
- **Scalability**: Our course enrollments are currently 72 per offering, but we will face pressure to increase enrollments.
- **Sustainability**: The course should not place undue burdens on teaching staff (instructors, grad TAs, and undergrad TAs).

## 3.1 Course Structure

The course is divided into two phases: a preparation (on-boarding) phase (typically 6-8 weeks), and a legacy code project phase (typically 2-4 weeks).

*3.1.1 On-boarding Phase.* The On-boarding phase is used to introduce core software engineering concepts such as testing and code coverage, Agile methodologies, and product management, as well as technical skills for developing web applications, including Java and React fundamentals. During the on-boarding phase, students work in the same teams that they will work in during the project phase; the difference is that during this phase, each team is working on the same assignment, and the assignments are more "exercises" in various aspects of backend and frontend web development (typically, specific skills in isolation) rather than authentic applications.

Even though the code itself may take the form of exercises, we nevertheless begin to gradually introduce aspects of the Agile methodology throughout the on-boarding phase, including standups,

[1] Though, after reading Richards' discussion of group size, it seems that even that also "depends".

the use of Kanban boards, the feature-branch/pull-request (PR) workflow, and retrospectives to reflect on team process.

Instruction during the on-boarding phase uses the flipped classroom model; lecture material is the form of either a mini-lecture at the beginning of a class or made available in a pre-recorded video. Most class meeting time is devoted to hands-on assignments carried out in a team context, so that students are able to practice their communication and collaboration skills.

When the course is offered face-to-face (which was true from 2010-2020, and most of the 2021-2022 academic year), we tried to ensure that the course was scheduled in a classroom in which it was possible for students to physically arrange themselves into groups. When teaching was fully remote or hybrid in 2020-2022, we used Zoom breakout rooms for team work, with one room per team, plus a number of unassigned "first-come first-served" breakout rooms; the latter were used when a team wanted to subdivide further in order to work on different tasks. Our most recent offerings have taken place in a classroom set up specifically for group work: each team sits around a U-shaped table with a large computer monitor on the wall at one end. This is particularly helpful for hybrid teams; they can display the Zoom room on the monitor.

*3.1.2 Project Phase.* The final phase of the course is the legacy code project phase. Course staff prepare either two or three legacy code projects, and for each project, they prepare two feature epics. Each course discussion section has 24 students—four teams of six students each. Pairs of teams (12 students) in the same discussion section are assigned a GitHub repository and pair of epics (groups of related issues) to code in that repository. Students are encouraged to communicate across teams within their section to coordinate feature development and remedy bugs and regressions. At least twice during the project phase, students engage in an Agile retrospective and agree on a specific process improvement that they will try to implement during the next sprint.

In previous iterations of the course, we constructed one unique epic per student team to mirror the structure of an actual software development organization. This hit a brick wall in the Winter 2020 offering of this course, when we had two 72-student lecture sections and 24 student teams. Although we did manage to create 24 unique epics, it required a level of effort on the part of the course staff that we recognized as unsustainable. In addition, merging code into the main branch became a huge bottleneck for the students.

To ensure that our course scales with increasing enrollment, we now create only two epics per project. While epics are unique within a discussion section, multiple teams in the course may be working on the same epic. This has three benefits: (1) Workload on course staff is reduced. (2) During final demos, students are able to see how different teams approach the same problem. (3) After the course is completed, staff can compare each implementation and construct a composite code base with the best designs as the one that continues into the next course iteration.

## 3.2 Project Bootstrapping and Curation

When selecting ideas for legacy code projects, we choose ideas that are sufficiently functionally complex that students will be able to continually add new features over multiple years. We focus on projects where students are the target users; this allows students to play the role of the customer to contribute new ideas for future iterations of the product.

Once an idea is selected, course staff start by building a separate minimum viable product (MVP) using the desired tech stack. This MVP is a bare-bones application that provides the absolute minimal amount of code needed to demonstrate the stack, while maintaining a clear, extensible structure for project-specific functionality. As our projects are web applications, our MVPs include features such as OAuth authentication, logging of users in an SQL database, and scaffolding for getting resources from external APIs. Recent project MVPs have also included support for component / API documentation plugins, as well as background jobs to allow for more complex features. Prior work by Chow et. al. [6] has shown that students are far more likely to adopt maintenance practices, such as complete unit test coverage, if the code base starts in a good state; accordingly, our MVPs include open-source unit testing, test coverage, mutation testing and linting frameworks. The code bases we present to students as starting points achieve 100% code coverage and mutation testing scores, which students are expected to maintain as they make changes.

## 3.3 Student Teams

Students are assigned to development teams of at most six students, and they work in these teams throughout the course. We attempt to finalize team assignments *before the first course meeting* so students can start team activities on day one. We utilize the CATME TeamMaker software [19] to assign students to teams based on a pre-survey completed by each student. Criteria used within the pre-survey include gender—prior work cited in the paper on CATME suggests avoiding forming teams where an individual is the only person of their gender—as well as students' primary operating system, preferred working style (in-person, hybrid, or online), and self-assessment of previous Java and React knowledge. Students are also given the opportunity to express a preference of who they want to work with—we try to accommodate these requests when they are mutual and satisfiable, but typically, these are with at most one other person. Thus, even in these cases, there is some degree to which students do not control the makeup of their teams.

Each student team is also assigned a staff mentor that is either a graduate or undergraduate teaching assistant. During the on-boarding phase, mentors act as the first point of staff contact for all questions from the team. In the project phase, mentors fulfill a more managerial role, answering design-related questions from students and performing code reviews of PRs.

Team performance over time is assessed using CATME Peer Evaluations, where students are given opportunities to anonymously evaluate the performance of other team members in five "teamwork dimensions" that reflect the soft skills used in effective student teams [24]. Intermediate peer evaluations conducted before the end of the course are purely informational—students can give and receive feedback without impacting a grade. However the final CATME peer evaluation can influence each team member's grade: an adjustment factor is determined based on the relative student performance within a team. This adjustment factor is used as a multiplicative factor in a student's final project grade.

### 3.4 Communication Tools

We use Slack as our primary communication tool within the class. In addition to special topic help channels, each team has one team channel that they can freely use to hold team discussions, as well as the ability to send direct messages (DMs) to other students and the course staff. After the first day of the course, all course-related announcements are made within the Slack to encourage students to continuously monitor the workspace for updates. Furthermore, we set up a dedicated help queue channel for use during instructional periods, where students can post a help request to request in-person assistance from course staff.

We encourage students to use the official class communications channel (Slack) rather than setting up their own side-channels (for example, GroupMe, Discord, etc.). We explain that when students use the official channel, staff are able to observe and offer help when we see that students are stuck on something where we may be able to offer a solution. However, this is only an "encouragement"—we recognize that we are unable to exert control here, and for that matter, micromanaging a team's decisions is contrary to the values of the Agile development process.

During fully-remote or hybrid instruction, to facilitate remote work, we use Zoom as our video communication platform. We specifically use the Breakout Rooms feature to give each team their own individual working space within a course instructional period, should teams have one or more members participating remotely.

### 3.5 Flipped Classroom

During both the on-boarding and project phases of the class, we use a flipped classroom model, with a specific strategy: we set aside one channel of the Slack workspace as a help-queue. There are existing tools for managing help queues such as [27], however we found that a Slack channel served our needs well. We mapped particular reaction emojis to whether a staff member was addressing an issue, whether it was complete, or whether a staff member needed assistance from another staff member. One nice feature was that we had the help-queue slack channel displayed on computer monitors throughout the classroom, which made it easy for course staff to see who needed help and to follow up. We were also able to use the Slack app on our cell-phones to monitor the queue, and take responsibility for help requests as they arose.

### 3.6 Tech Stack

Projects in this course focus on full-stack web development using the Model-View-Controller design pattern. Our project code bases consist of two main components: a Java Spring Boot backend packaged with Maven and a React frontend packaged with npm. These are both modern, open-source frameworks that are well-documented, used in many professional products, and have widespread community support on developer Q&A forums such as Stack Overflow. To assist with backend controller development, we use Swagger UI as a tool for debugging and documenting our Spring Boot API routes. Similarly, our frontend utilizes Storybook to design and document React components. Both of these documentation tools are able to be statically deployed, allowing for easy testing and collaboration between members. To assist with user interface (UI) development, we make use of the Bootstrap UI library to create a consistent user experience across platforms.

Testing is also a large aspect of this course. Our project code bases utilize JUnit and Jacoco for Java unit testing and coverage, and similarly, Jest for React component testing and coverage. Additionally, we utilize Codecov, an online code coverage solution integrated with GitHub that provides a unified view of code coverage that can be tracked over time and used to enforce coverage thresholds at the code review level. Furthermore, we utilize mutation testing to evaluate the quality of student-developed unit tests, specifically, Pitest for Java mutation coverage and Stryker Mutator for JavaScript mutation testing.

### 3.7 GitHub

The main development tool used within this course is GitHub. While GitHub's core functionality is to serve as a version control system for project code bases, GitHub has a number of integrated tools and project management features designed to facilitate the development workflow. Issues provides a platform for students to document new feature requests and bug reports from product management exercises. Staff members can further utilize issues to create feature epics, which are then assigned to teams in the legacy code phase. In this phase, teams can visualize and track issue progress in Projects, which provides a column-based project board similar to Kanban.

When code development is complete, we utilize Pull Requests (PRs) to facilitate the code review process. We require students to receive two approving code reviews before any change can be merged; for legacy code projects, one of these reviews must come from a member of the instructional staff. In addition to human review, we use GitHub's Continuous Integration/Continuous Delivery (CI/CD) system, GitHub Actions, to run the project's unit test suites and upload results to Codecov, which will enforce complete coverage through a status check. We additionally utilize Actions to build and deploy a static Storybook to GitHub Pages. Finally, we utilize branch protection rules to restrict student push access to the primary branch, which enforces a staff review on any student pull requests. This gives an opportunity for staff to evaluate every student pull request for code correctness and ensure professional development etiquette is followed.

### 3.8 Deployment

Projects are deployed using Heroku, a cloud platform for deploying web applications. For each project, we maintain at least two Heroku applications - one production deployment and at least one quality assurance (QA) deployment. As more than one team is assigned to each project in the legacy code phase of the course, every team will share one single production deployment managed by the course staff, while also maintaining individual team QA deployments. The production deployment reflects the latest tested and approved code set to automatically deploy from a repository's primary branch. The QA deployment is a production-like test environment that can be used to validate changes in the code review phase. These two phases create a simple development life cycle that can facilitate feature development across multiple teams.

## 4 BRIDGING THE GAP

We use the six themes presented in Craig et. al. [9] as a framework for discussing the gap between existing CS pedagogy and professional development practices. This paper builds on previous literature outlining common learning barriers for recent graduates by stating incoming software developers' shortcomings in terms of specific differences between academia and industry.

### 4.1 What: Well-defined vs. Open-ended Scope

Academic assignments in CS courses are often accompanied by a strict set of requirements to be followed, whereas professional projects are often ambiguous, with open-ended feature requests. Our course aims to provide students with an experience of this ambiguity through a combination of issue grooming exercises and product management/ownership activities.

Students are first introduced to open-ended issues through a product management activity where they are shown a deployed version of the project they will inherit in the project phase. As students are the target users of our legacy code applications, they are asked to use the app as a normal user and note any issues encountered or features that would make the application more valuable. Students are then instructed to share their individual feedback within their team and consolidate ideas into one feature document.

These ideas are then aggregated by the course staff into several feature epics, which form the basis of the product backlogs assigned to the students in the project phase. The product backlogs written by the staff typically contain a mix of issues that have been well-groomed with specific acceptance criteria (as examples, and to help students get started quickly), along with higher-level issues that are more open-ended and deliberately vague. Students must then refine the issue into a specific user story with clear acceptance criteria, while taking into consideration any limitations of the existing codebase. Through this process, students will come across multiple ways to approach a feature and will have to juggle design considerations and their trade-offs. This may involve a further dive into the code base or an architectural discussion with a member of the course staff.

### 4.2 When: Short vs. Long Time Span

Academic projects often follow a "one-and-done" approach; they have a lifespan of a few days or weeks, and upon completion of the course, they are rarely ever looked at again. Many existing courses have attempted to address this issue by conducting term-length (10-15 weeks) greenfield projects, but these are still considerably shorter than industry projects which often span multiple years. By contrast, our course project code bases are maintained across multiple successive iterations of the course.

An initial attempt at establishing such a course structure was to conduct one initial quarter of student-designed greenfield projects, and then maintain those as brownfield projects in future quarters. We found that student working on their first large-scale project often did not have the necessary design skills to produce a code base that was maintainable, or to design a product that would be sufficiently complex to provide a challenge over multiple quarters.

Our current approach, when introducing new legacy code projects, is the course staff (instructor and TAs) designs and implements an MVP with plans for long-term maintenance built-in, including code structure, complete testing, linting, and documentation, as documented in Section 3.2. This initial MVP becomes the basis for the first iteration of student work; thus, even from the first course iteration, students are working with a legacy code base (albeit a smaller one than in future iterations). Since staff are the initial greenfield developers, staff are able to create the realistic conditions of working with a large, matured code base that has constraints on design and guide students in maintaining these constraints.

### 4.3 Who: Small Groups vs. Large Teams

Academic greenfield projects are often worked on individually or in small groups of 4-5 students. However, real-world projects are often carried out by teams that share a code base with tens or hundreds of other developers. In our legacy-code based course, students have the experience of working in a single repository that is shared across two teams that include 10-12 students. This provides students with additional experience with the need to communicate across teams about architectural decisions, and to work to avoid merge conflicts.

### 4.4 Why: Learning vs. User Needs

Projects in academic courses are typically used to gain hands-on experience with a particular foundational concept in computer science. This marks a sharp contrast from industry applications, which are always designed around addressing specific user needs, which will constantly evolve over time.

We provide students experience with addressing user needs by intentionally selecting projects where students are the intended users. Two examples are (1) a web application for searching the course offerings of our institution over time, in ways that official course searches cannot provide, and (2) a simulation game used in a course in Environmental Chemistry. This allows students to act as both the customer and the product owner.

### 4.5 How: Ad-hoc vs. Professional Tools

Because academic projects are often "greenfield" without a need for long-term maintenance, students are only incentivized to use the minimal number of tools needed complete the task at hand. Many courses encourage the use of GitHub, testing, and test-driven development (TDD), but usage of such tools is often very shallow (e.g., GitHub is often limited to pushing and pulling) and methodologies are challenging to enforce at an individual level.

We resolve this by incorporating a suite of professional tools to aid the development of legacy code projects. When choosing our project tech stack, we look for modern and open-source frameworks and tools with documented usage in industry. GitHub is taught not just as a place for housing code; we also take advantage of branching, issues, pull requests, and a CI/CD pipeline through GitHub Actions. Furthermore, testing is consistently emphasized as an integral part of the development process. As discussed earlier, we incrementally introduce agile practices throughout the on-boarding phase, so that by the time the project phase begins, students are familiar with standups, issues, Kanban boards, code reviews, and retrospectives.

### 4.6 How Big: Small vs. Large Systems

As most academic projects are small greenfield code bases that are worked on in small groups, students often follow a mindset that they must understand all code that they write. Because such code bases have no intention to be shared, they often lack professional styling and structure, and authors have little incentive to perform refactors, practice good design patterns, and write documentation. These projects are a sharp contrast from industry code bases, which are often large in size, iterated on by multiple teams in parallel, and often include code written by developers who have since left the organization. This often leads to young developers lacking the skills to read code and start making contributions in industry positions.

This theme serves as the largest motivator for the use of brownfield legacy code projects. By creating projects that are designed to be iterated on with each successive course, students gain experience working in code bases where they don't have full ownership. There is an expectation for code to be well-maintained through code styling and thorough testing. Additionally, students can begin to appreciate the need to adhere to the existing coding standards, styles, and patterns in the code base so as to keep it maintainable for future developers.

Students are also forced to acknowledge that it is often not possible to deeply understand the entire code base and all of its dependencies in detail; abstraction is essential to success (as in real-world software engineering).

## 5 REFLECTION

Course evaluations show that, overall, students are satisfied with the course; in narrative evaluations, they frequently mention they appreciate the relevance of the skills and knowledge in the course to their future work in the software industry. The main critiques are that (1) the course is often not as organized as students would prefer; this reflects the fact that this course has been a constant "work-in-progress" over the past ten years, (2) the students wish they had more time for the project phase of the course.

From the point of view of the instructional staff, there have been several challenges:

### 5.1 Keeping up with the pace of change in professional software tools

Each year, there are new versions of Spring Boot, React, Bootstrap, Java itself, as well as dozens of other tools and systems on which the course depends. This requires updates to course materials and code. In addition, over the last decade, practices in software development in general, and web development in particular, have changed rapidly.

### 5.2 Working towards a scalable and sustainable course structure

This course has been a labor of love for the lead author, and as such they have been willing to put many extra hours into the design and implementation of the course. However, in the long-term, it is not sustainable to put twice as much effort into a particular course as other courses, nor it is reasonable to expect this level of effort from colleagues that may teach the course in the future. The course is now manageable with 72 students, but due to enrollment pressures, we will need to scale to many more students in the future.

### 5.3 Assessment of student projects

Our approach to assessment of the team projects is one we have not seen elsewhere. The team is responsible, over the course of the project, for completing 100 points worth of work. They are given a "backlog" of issues (in the sense used in the Agile software development methodology [1]). Each time a member of the team submits a PR to address one of these issues, it is code reviewed, first by a peer team member, and then by a member of the course staff. When any/all issues raised in the code review are addressed, the PR is merged into the main branch, and points are awarded: 5, 10, or 20. In this way, the team works towards their project grade. It is feasible for each team to reach 100 points, provided that they plan their work well, and are able to solve the issues presented to them. Teams are also encouraged to propose their own issues, or create issues for bugs they find.

One advantage of this approach is that it allows for feedback and revision, which is often not the case in other computing courses. A second advantage is that it is authentic, given that code review is often a feature of real-world software development environments.

One aspect of this method of assessment may be controversial: our experience has been that nearly all teams routinely put in the work necessary to reach the grade of 100%. Some seem to reach that milestone more easily and quickly than others, but they all eventually get there. Whether this is a feature or a bug is a matter of educational philosophy about which instructors may have reasonable disagreements.

### 5.4 Small, incremental PRs

It is good practice to make small, incremental commits and small, incremental pull requests. Regrettably, we have found that some student teams have a tendency to make large, monolithic commits and PRs. This results in several problems: (1) These are much harder to code review. (2) These are much more likely to result in merge conflicts. (3) The merge conflicts that result are likely to be more complex and difficult to resolve. (4) It is much more likely that the various individuals or teams working on the code base will get out-of-sync with one another in terms of design decisions. All of these things are also true of real-world software engineering practice, so this is a valuable lesson for the students to learn. However, students and instructors alike tend to be happier when this lesson is not learned through direct experience of the downsides of the wrong choice.

## 6 SUMMARY

In this paper we have described the design of an undergraduate software engineering course based on legacy code. We briefly surveyed the literature on the gap between students' preparation and the skills/knowledge needed for success in the software industry, and described how this course design can help to bridge that gap.

## ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (United States), Award number: 1915198

# REFERENCES


[1] Agile Alliance. 2022. What is a Backlog? https://www.agilealliance.org/glossary/backlog

[2] Andrew Begel and Beth Simon. 2008. Novice software developers, all over again. In *Proceedings of the Fourth international Workshop on Computing Education Research (ICER '08)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1404520.1404522

[3] Andrew Begel and Beth Simon. 2008. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 226–230. https://doi.org/10.1145/1352135.1352218

[4] Grant Braught, John Maccormick, James Bowring, Quinn Burke, Barbara Cutler, David Goldschmidt, Mukkai Krishnamoorthy, Wesley Turner, Steven Huss-Lederman, and Bonnie Mackellar. 2018. A multi-institutional perspective on H/FOSS projects in the computing curriculum. *ACM Transactions on Computing Education (TOCE)* 18, 2 (2018), 1–31. Publisher: ACM New York, NY, USA.

[5] Gail Carmichael, Christine Jordan, Andrea Ross, and Alison Evans Adnani. 2018. Curriculum-Aligned Work-Integrated Learning: A New Kind of Industry-Academic Degree Partnership. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 586–591. https://doi.org/10.1145/3159450.3159543

[6] Scott P. Chow, Tanay Komarlu, and Phillip T. Conrad. 2021. Teaching Testing with Modern Technology Stacks in Undergraduate Software Engineering Courses. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 241–247. https://doi.org/10.1145/3430665.3456352

[7] Ben Coleman and Matthew Lang. 2012. Collaboration across the Curriculum: A Disciplined Approach to Developing Team Skills. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 277–282. https://doi.org/10.1145/2157136.2157220

[8] Edward J Coyle, Leah H Jamieson, William C Oakes, et al. 2005. EPICS: Engineering projects in community service. *International journal of engineering education* 21, 1 (2005), 139–150.

[9] Michelle Craig, Phill Conrad, Dylan Lynch, Natasha Lee, and Laura Anthony. 2018. Listening to early career software developers. *Journal of Computing Sciences in Colleges* 33, 4 (4 2018), 138–149.

[10] M. Exter. 2014. Comparing educational experiences and on-the-job needs of educational software designers. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 355–360. https://doi.org/10.1145/2538862.2538970

[11] Vivienne Farrell, Graham Farrell, Paul Kindler, Gilbert Ravalli, and David Hall. 2013. Capstone project online assessment tool without the paper work. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education (ITiCSE '13)*. Association for Computing Machinery, New York, NY, USA, 201–206. https://doi.org/10.1145/2462476.2462484

[12] Vivienne Farrell, Gilbert Ravalli, Graham Farrell, Paul Kindler, and David Hall. 2012. Capstone project: fair, just and accountable assessment. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 168–173. https://doi.org/10.1145/2325296.2325339

[13] Wouter Groeneveld, Brett A. Becker, and Joost Vennekens. 2020. Soft Skills: What Do Computing Program Syllabi Reveal About Non-Technical Expectations of Undergraduate Students?. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 287–293. https://doi.org/10.1145/3341525.3387396

[14] Wouter Groeneveld, Hans Jacobs, Joost Vennekens, and Kris Aerts. 2020. Non-Cognitive Abilities of Exceptional Software Engineers: A Delphi Study. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1096–1102. https://doi.org/10.1145/3328778.3366811

[15] Wouter Groeneveld, Joost Vennekens, and Kris Aerts. 2019. Software engineering education beyond the technical: A systematic literature review. In *Proceedings of the 47th Annual SEFI Conference*. SEFI, Budapest, Hungary, 1607–1622.

[16] Wouter Groeneveld, Joost Vennekens, and Kris Aerts. 2021. Identifying Non-Technical Skill Gaps in Software Engineering Education: What Experts Expect But Students Don't Learn. *ACM Trans. Comput. Educ.* 22, 1, Article 1 (oct 2021), 21 pages. https://doi.org/10.1145/3464431

[17] Reid Holmes, Meghan Allen, and Michelle Craig. 2018. Dimensions of Experientialism for Software Engineering Education. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training* (Gothenburg, Sweden) *(ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 31–39. https://doi.org/10.1145/3183377.3183380

[18] Christopher Hundhausen, Adam Carter, Phillip Conrad, Ahsun Tariq, and Olusola Adesope. 2021. Evaluating Commit, Issue and Product Quality in Team Software Development Projects. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 108–114. https://doi.org/10.1145/3408877.3432362

[19] Richard Layton, M. Ohland, and H. Pomeranz. 2007. Software for Student Team Formation and Peer Evaluation: CATME Incorporates Team-Maker. In *2007 ASEE Annual Conference & Exposition*. ASEE, Honolulu, HI, 397–402. https://doi.org/10.18260/1-2--2355

[20] Marcia A Mardis, Jinxuan Ma, Faye R Jones, Chandrahasa R Ambavarapu, Heather M Kelleher, Laura I Spears, and Charles R McClure. 2018. Assessing alignment between information technology educational opportunities, professional requirements, and industry demands. *Education and Information Technologies* 23, 4 (2018), 1547–1584.

[21] Ralph Morelli, Allen Tucker, Norman Danner, Trishan R. De Lanerolle, Heidi J. C. Ellis, Ozgur Izmirli, Danny Krizanc, and Gary Parker. 2009. Revitalizing Computing Education through Free and Open Source Software for Humanity. *Commun. ACM* 52, 8 (aug 2009), 67–75. https://doi.org/10.1145/1536616.1536635

[22] Debora Maria Coelho Nascimento, Christina von Flach Garcia Chavez, and Roberto Almeida Bittencourt. 2019. Does FLOSS in Software Engineering Education Narrow the Theory-Practice Gap? A Study Grounded on Students' Perception. In *Open Source Systems*, Francis Bordeleau, Alberto Sillitti, Paulo Meirelles, and Valentina Lenarduzzi (Eds.). Springer International Publishing, Cham, 153–164.

[23] Tom Nurkkala and Stefan Brandle. 2011. Software Studio: Teaching Professional Software Engineering. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 153–158. https://doi.org/10.1145/1953163.1953209

[24] Matthew W. Ohland, Misty L. Loughry, David J. Woehr, Lisa G. Bullard, Richard M. Felder, Cynthia J. Finelli, Richard A. Layton, Hal R. Pomeranz, and Douglas G. Schmucker. 2012. The comprehensive assessment of team member effectiveness: Development of a behaviorally anchored rating scale for self-and peer evaluation. *Academy of Management Learning & Education* 11, 4 (2012), 609–630. Publisher: Academy of Management Briarcliff Manor, NY.

[25] A. Radermacher and G. Walia. 2013. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 525–530. https://doi.org/10.1145/2445196.2445351

[26] Debbie Richards. 2009. Designing Project-Based Courses with a Focus on Group Formation and Assessment. *ACM Transactions on Computing Education* 9, 1 (March 2009), 2:1–2:40. https://doi.org/10.1145/1513593.1513595

[27] Aaron J. Smith, Kristy Elizabeth Boyer, Jeffrey Forbes, Sarah Heckman, and Ketan Mayer-Patel. 2017. My Digital Hand: A Tool for Scaling Up One-to-One Peer Teaching in Support of Computer Science Learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 549–554. https://doi.org/10.1145/3017680.3017800

[28] Anna Stepanova, Alexis Weaver, Joanna Lahey, Gerianne Alexander, and Tracy Hammond. 2021. Hiring CS Graduates: What We Learned from Employers. *ACM Trans. Comput. Educ.* 22, 1, Article 5 (oct 2021), 20 pages. https://doi.org/10.1145/3474623

[29] Sander Valstar, Caroline Sih, Sophia Krause-Levy, Leo Porter, and William G. Griswold. 2020. A Quantitative Study of Faculty Views on the Goals of an Undergraduate CS Program and Preparing Students for Industry. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 113–123. https://doi.org/10.1145/3372782.3406277

[30] Maria Vasilevskaya, David Broman, and Kristian Sandahl. 2015. Assessing Large-Project Courses: Model, Activities, and Lessons Learned. *ACM Trans. Comput. Educ.* 15, 4, Article 20 (dec 2015), 30 pages. https://doi.org/10.1145/2732156

[31] Brian R. von Konsky and Jim Ivins. 2008. Assessing the capability and maturity of capstone software engineering projects. In *Proceedings of the tenth conference on Australasian computing education - Volume 78 (ACE '08)*. Australian Computer Society, Inc., AUS, 171–180.