Design and Evaluation of an "Athletic" Approach to Software Engineering Education

PHILIP JOHNSON, University of Hawaii, USA

Modern web application development provides an attractive application area for introductory software engineering education, as students have direct experience with the domain and it provides them with the potential to gain practical, real-world skills. Achieving this potential requires the development of competency with a multiple component tech stack for web application development, which is challenging to acquire within a single semester. In this research, we designed, implemented, and evaluated a new pedagogy called "athletic software engineering" which is intended to help students efficiently and effectively acquire competency with a multiple component tech stack as a precursor to a web application development project. We evaluated the pedagogy over 4 years and six semesters with 286 students and found strong evidence for its effectiveness.

CCS Concepts: • **Social and professional topics → Software engineering education**; • **Applied computing → Collaborative learning**; • **Software and its engineering →** *Software design engineering*; *Software design techniques*; *Programming teams*;

Additional Key Words and Phrases: Software engineering education

ACM Reference format:
Philip Johnson. 2019. Design and Evaluation of an "Athletic" Approach to Software Engineering Education. *ACM Trans. Comput. Educ.* 19, 4, Article 41 (August 2019), 25 pages.
https://doi.org/10.1145/3344273

1 INTRODUCTION

The rise of "tech stacks" for modern application development has produced a quandry for software engineering educators. On the one hand, competency with a tech stack makes it possible for students to develop and deploy modern, "professional" applications in a relatively short period of time, which in turn makes it more possible than ever before for students to experience "real-world" software engineering issues in the classroom. On the other hand, developing tech stack competency is a non-trivial undertaking, and it is increasingly possible for students to become mired in low-level tech stack issues that prevent them from engaging with higher-level software engineering concepts and ideas.

This article presents the design and evaluation of a new pedagogy called Athletic Software Engineering, which is intended to enable students to efficiently acquire tech stack competency to improve their overall experience of software engineering education. Our approach divides the semester into two parts. The first, "athletic-oriented" part focuses on helping students obtain fluency

Authors' address: P. Johnson, University of Hawaii, Information and Computer Sciences, 1680 East-West Road, Honolulu, HI, 96812; email: johnson@hawaii.edu.

1946-6226/2019/08-ART41 $15.00
https://doi.org/10.1145/3344273

with the "mechanics" of a tech stack. The second, "project-oriented" part enables to students to apply their new-found skills to the design and implementation of a web application.

The athletic portion of the semester introduces several novel pedagogical features. As will be discussed further, the athletic curriculum is structured as a sequence of skills to be mastered, not concepts to be memorized. Each skill is documented by a set of practice exercises, called a "Workout of the Day (WOD)." Each practice WOD includes a video that demonstrates both how to solve the problem, and the time required to solve it once competency with the skill is acquired. Students are instructed to repeat the practice WODs until competency is achieved, which is operationalized as being able to solve the exercise in a specific length of time. Finally, competency with each skill is assessed and graded through an in-class WOD, similar to the practice WOD, in which students must solve the problem both correctly and within a time limit to receive credit.

To evaluate this approach, we gathered data from 4 years of use across six semesters and 286 students. Data sources included the following: a custom mid-semester questionnaire, the "WOD cards" that record the results of weekly in class WODs, self-reported student data on the number of times they repeated a practice WOD, as well as the standard end-of-semester institutional course evaluation survey. We used this data to investigate four research questions: (1) Is Athletic Software Engineering an effective pedagogy for learning software engineering?; (2) Do students comply with the basic components of Athletic Software Engineering?; (3) Does Athletic Software Engineering have side effects, such as creating competition, improving confidence, managing pressure, and improving focus?; and (4) How can Athletic Software Engineering be improved?

Our results provide strong evidence that Athletic Software Engineering is an effective pedagogy: On average, over 4 years, 88% of students preferred it to the more traditional, lecture-based pedagogies they experienced in their prior computer science classes. The results also provide evidence that many students repeat the practice WODs, that most find the in-class WOD with its all-or-nothing grading scheme to be helpful for learning, that the approach can create competitive feelings, that it can increase confidence in ones software development skills, and that it helps many students feel more comfortable programming under pressure.

The remainder of this article is structured as follows. Section 2 presents research related to this approach. Section 3 describes Athletic Software Engineering as we currently practice it. Section 4 describes the survey results and Section 5 their limitations. Section 6 discusses future directions for this pedagogy.

2 RELATED WORK

Software engineering educators have experimented with a wide variety of pedagogies over the past 40 years. From the inception of software engineering in the late 1970s up until the 1990s, the standard approach relied on a textbook such as Roger Pressman's "Software Engineering: A Practitioner's Approach" [32]. Instructors lectured about standard software engineering topics ("design," "testing," and "software review"), each covered by a separate chapter in the book. Evaluation involved classroom tests in which students recalled facts and terminology ("what is the difference between white box and black box testing"). Some of these textbook-oriented courses also included a small final project involving some programming in an attempt to provide "practical experience" with the lecture topics.

By the early 2000s, the problems with applying a traditional, textbook-oriented pedagogy to software engineering were well known [35]. Abstract understanding of terminology and concepts does not necessarily translate into practice. Fortunately, the rise of the Internet, World Wide Web, and configuration management systems like CVS and SVN significantly lowered the barriers to development and deployment of software and made possible a more active, collaborative, project-oriented software engineering pedagogy. Many different kinds of project-oriented pedagogies have been explored for software engineering education, including contributions to open source systems [11, 29, 31], collaboration with community and industry partners [6, 8, 10, 33, 34, 39], larger project scopes [9], vertical integration throughout the curriculum [16], simulation [3, 30], and a focus on design rather than implementation [13, 21].

Another alternative to textbook, lecture-based pedagogy that arose during this time was the application of studio-based learning principles common in disciplines like architecture to software engineering [7, 23, 27]. In studio-based learning, there is a physical location (the studio) where students congregate to work, there is ongoing feedback and criticism from instructors (and potentially other students) on the evolving project, and an emphasis on rapid iteration of design solutions. Studio-based learning improves on traditional textbook-oriented pedagogy by increasing engagement and emphasizing concrete work products rather than conceptual knowledge acquisition.

Starting around 2010, YouTube began allowing free uploads of videos longer than 10 minutes, eliminating a crucial technical and economic barrier to the use of "flipped" or "inverted" classrooms, and this approach has also been explored in software engineering education [12, 15, 28]. In a nutshell, flipped classrooms invert the traditional approach in which class periods focus on (passive) lectures and homework focuses on (active) problem solving. Instead, YouTube or other online media enable students to watch the lectures as "homework," leaving in-class time available for problem solving. Just like project and studio-based learning, a flipped classroom has the potential to increase student engagement by increasing the availability of students and instructors for collaborative learning and problem solving.

It is useful to note that the project, studio, and flipped pedagogies are not mutually exclusive, and many hybrids can be created by combining them. A studio-based approach can involve "flipping," and a flipped classroom might or might not involve a project orientation.

The past 10 years has included yet another technical breakthrough with an impact on software engineering pedagogy: the maturation of web application frameworks including Ruby on Rails, Django, Laravel, and Spring. These frameworks make it possible for software engineering classroom projects to involve the design, implementation, and deployment of cloud-based multi-user cross-platform applications. These kinds of projects are tremendously incentivizing to students, since they result in "real-world" systems and also imbue students with a marketable skillset. A major problem with integrating web application development into a software engineering pedagogy (be it project, studio, or flipped) is the large number of components comprising the minimal "tech stack" necessary for students to experience modern software engineering practices, as shown in Table 1.

To appreciate the magnitude of this issue, consider that a reasonable "tech stack" for software engineering classroom projects in the 1990s had three components: an editor (such as vim or emacs), a programming language (such as C++ or Java), and a library or two (such as Java's Swing toolkit for UI development). In contrast, Table 1 illustrates 10 components comprising a basic tech stack for web application development. Note that this set of components is not even complete: a tech stack for modern web application development will typically include additional components such as containerization, continuous integration, cross-platform compatibility testing, load testing, and requirements analysis. Sites such as stackshare.io provide details on modern tech stacks used for professional development.

At our institution, our undergraduate software engineering class shifted in 2011 to a focus on web application development as the target domain. We discovered that the conceptual benefits of this target domain could not be realized for any software engineering pedagogy (project, studio, or flipped) without understanding and addressing the challenge of *learning the tech stack*.

Table 1. Ten Tech Stack Components and Example Technologies for Web Application Development

Tech Stack Component	**Example technologies**
Programming Language	*Javascript*, Python, Java, Scala, Swift
Configuration Management	*git, GitHub Desktop*, SVN, Perforce
Project Hosting	*GitHub*, GitLab, Sourceforge
Development Environment	*IntelliJ IDEA*, Eclipse, XCode, Visual Studio
Quality Assurance	*ESLint*, PyLint, FxCop; *Mocha, Chai*, JUnit, MSTest
User Interface	*HTML, CSS, Semantic UI, React*, Bootstrap, Foundation
Database	*MongoDB*, MySQL, Postgres
Web Application Framework	*Meteor*, Play Framework, Laravel, Spring ASP.Net
Project Management	*GitHub Projects*, Jira, Asana, Trello
Deployment	*Galaxy*, Heroku, Google App Engine

Tech stack components used in this research project are shown in *italics*.

Consider the following development scenario, which is typical during project development:

> A student chooses a development task from the GitHub Project board, and moves it from "Backlog" to "In Progress". She then creates and publishes a new branch off the project's master branch in which to accomplish the task. The task involves the creation of a new page containing a form in the application. Using IntelliJ IDEA, she writes a new React component in Meteor along with a route to that page, several Javascript callback functions to manage form processing, and a responsive layout for the page using HTML, CSS, and Semantic UI. To store the results from the form, she extends a MongoDB collection with new fields, and writes a test using mocha and chai. During implementation, she notes any ESLint code style errors as they occur and removes them. When the page works correctly, she commits her branch to GitHub, merges the results into the master branch, and moves the task from "In Progress" to "Done". Finally, she deploys the improved implementation in the master branch to Galaxy so that she can receive feedback about it from her users.

Notice that this simple development scenario nevertheless requires interaction with 15 technologies spanning all 10 components of the tech stack: Javascript, git, IntelliJ Idea, ESLint, Mocha, Chai, GitHub, HTML, CSS, Semantic UI, React, MongoDB, Meteor, GitHub Projects, and Galaxy. Beyond the technology implications, this scenario also requires the student to address interesting design problems (What information is being manipulated on that page? Is that the right information to manipulate? Is the chosen page the appropriate place to manipulate that information?) as well as process problems (Is this the appropriate task for her to work on right now? How might that task interact with others being worked on by others in the group? What communication should be occurring to make sure her teammates do not implement conflicting code?)

A key insight from our early attempts to incorporate modern web application development into our one-semester software engineering course is this: unless students are competent with the dozen or more technologies in the stack, it is difficult for them to actively engage with higher-level "design" and "process" problems. For example, if a student does not understand how to implement a form in Semantic UI, then that becomes their focus of attention; students will not, in practice, invest time in comparing and contrasting form *design* alternatives if they do not understand how to *implement* any of them.

Table 2. Mechanics vs. Creativity for Selected Technologies

Tech	Mechanics	Creativity
Javascript	How to use reduce()?	When to employ reduce() (and other functional constructs)?
git	How to create and merge branches?	When is it appropriate to create or merge a branch?
mocha, chai	How to create and run test cases?	What needs to be tested? When should test cases be run?
Semantic UI	How to create a nav bar?	What operations should be available from the navbar?
MongoDB	How to create and retrieve a collection of documents?	What collections should be created? How are documents related?
GitHub Projects	How to create a task board?	How to design tasks and assign them to enable concurrent development?

Our approach to addressing this problem involves a recognition that in modern web application development, learning the "mechanics" of the technologies in the underlying tech stack is a significant activity. By "mechanics," we mean the basic, domain-independent operation and/or use of constructs, which contrasts with "creative" or "design" intensive activities, in which the technologies are applied and combined in a unique way to solve a specific problem in a specific domain. Table 2 illustrates this difference for a few technologies.

When the tech stack for a class involves only one or two technologies, then the overhead of learning the "mechanics" is relatively small and can become an implicit part of learning the higher-level software engineering skills. What we discovered when we shifted to web application development is that expecting "implicit" learning of the mechanics for all levels of a sufficiently complex tech stack creates significant, and sometimes unsurmountable barriers to practicing the higher-level software engineering skills that involve "creativity." Lack of competency with the mechanics appeared to result in low quality final projects whose design and implementation revealed a struggle just to implement any kind of web application at all within a single semester.

We do not feel the solution to this problem is to simplify the tech stack: it is already simplified from those used in industry, and it is unclear how to significantly reduce the tech stack without losing coverage of fundamental aspects of modern software engineering. One could also abandon the domain of web application development altogether, but many other domains have equally complicated tech stacks.

Instead, we drew inspiration from two athletic endeavors with which we had experience: outrigger canoe paddling and CrossFit (although our observations are certainly not unique to these disciplines). In both paddling and CrossFit, there is explicit instruction regarding the low-level "mechanics" of the physical activities, as well as the high-level, "strategic" activities. Low-level mechanics are practiced and reinforced continuously, not just by beginners, but also by experts. Finally, in both of these endeavors, mastery of low-level mechanics is manifested by *speed*. It is important to note that in athletic endeavors, speed is not an outcome or a reflection of sloppiness in execution or a tradeoff in quality. Instead, speed is a natural outcome of *fluency* and *mastery*. Speed reflects a lack of sloppiness and a commitment to quality in mechanics and execution.

We hypothesized that an effective way to teach students software engineering in the context of a complicated tech stack would be to teach the mechanics separately and explicitly, emulating the way coaches teach mechanics in sporting activities like CrossFit. We could assess mastery of the mechanics through timed exercises in which students would not only have to apply the mechanics correctly but also efficiently. After students develop mastery of the mechanics, we could move on to "creative" activities, in which time-to-completion no longer makes sense as an evaluation criteria.

We have been experimenting and refining this pedagogy, which we call "Athletic Software Engineering," over the past 5 years. Athletic Software Engineering splits our one-semester software engineering course into two distinct parts. The first part focuses on developing mastery of the low-level mechanics associated with each component of the tech stack. During this "athletic" part, we measure their mastery of mechanics by their ability to perform basic operations with each component of the tech stack both correctly and quickly. The second part focuses on design and implementation of a final project. The hope is that having mastered the mechanics of the tech stack, students will now spend most of their time and energy on "creative" aspects of the problem such as design and performance. During the second, non-athletic part of the course, we do not explicitly evaluate them on the time taken to design but rather on the state of the final project at its completion.

It is important to emphasize that the time-constrained nature of athletic software engineering assessments does not incentivize or reward sloppiness and low quality but rather the opposite: a commitment to practice until the fundamental mechanics are mastered.

We published initial evaluations of our technique in 2016 [18, 20] after 2 years of use. This article presents results from our experiences and data collection over a total of 4 years and provides a more comprehensive evaluation.

Software Engineering is a vast area, and any one-semester course must make choices about what material to include and what material to exclude [38]. Our curriculum mostly follows an approach to acquiring "agile" competencies as discussed in References [22, 25]. From this perspective, the initial athletic portion concentrates on agile engineering practices and the subsequent project portion concentrates on agile management practices. It is also somewhat related to the Practice Based Studio approach at Carnegie Mellon [24], although the practice areas are simplified, our approach is more prescriptive, and the instructor serves as the sole mentor.

Athletic software engineering has similarities to Active Learning pedagogies [4, 5] in that it is organized in a way to promote constant challenge and engagement with the learning process as opposed to passive listening. It contrasts with active learning approaches, such as Reference [14], in that it does not require students to prepare lesson plans or videos as part of the pedagogy.

It also shares some similarities with spaced repetition [17], a learning technique in which increasing intervals of time between review of previously learned material has been shown to increase the rate of learning. The design of the Athletic Software Engineering curriculum does lead to intermittent review of previously learned material, since modules are cumulative and students need to apply concepts learned in prior modules to accomplish the tasks in subsequent modules. However, spaced repetition techniques involve testing the user on the subject matter and then adjusting the interval of time before the next test on the material based upon whether the user could successfully recall the material. In the current design of Athletic Software Engineering, the pedagogy does not adapt to the level of recall of the users; everyone simply gets re-tested on earlier material as a natural part of subsequent modules.

Bloom's Taxonomy [2] is a seminal classification scheme for learning outcomes, with interesting relationships to Athletic Software Engineering. Of interest to software engineering education is his approach to the "cognitive" domain, which views student learning as a sequence of six increasingly sophisticated cognitive tasks:

(1) *Knowledge Level*: The student is able to recall facts, basic concepts, and terms on a given topic.
(2) *Comprehension Level*: The student has to be able to organize, compare, translate, interpret, give descriptions, and state the overall idea related to a given topic.
(3) *Application Level*: The student is able to apply the retrieved knowledge in different ways to solve a given problem.

(4) *Analysis Level*: The student is able to identify motives and causes regarding a given topic, so that generalizations could be supported.
(5) *Synthesis Level*: The student is able to identify patterns from different elements or combine different parts to compose a whole.
(6) *Evaluation Level*: The student is able to make and defend judgements based on internal evidence or external criteria.

Bloom's Taxonomy has been applied in a variety of ways to software engineering education. Niazi [26] used it to design a master's-level course on software engineering. He found difficulties going beyond the Comprehension Level. Thompson [36] used Bloom's Taxonomy to assess exam questions from six institutions teaching first-year programming courses. Their results indicate that Bloom's Taxonomy can be a useful aid in assessment but that the cognitive skill (i.e., Level) required to answer an exam question may vary from individual to individual. Walker [40] used it to design and assess a lab-based programming course. Walker found that higher levels of the taxonomy could be engaged by requiring students to interpret and critique sample code.

Bloom's Taxonomy provides a useful way of thinking about the relative contributions of the "athletic" portion of the course that concentrates on the "mechanics" and the project-oriented portion of the course that concentrates on the "creative" aspects. The "mechanics" portion maps well to the first three levels of Bloom's taxonomy: Knowledge, Comprehension, and Application, while the "creative" portion maps well to the second three levels: Analysis, Synthesis, and Evaluation. That said, in this course, the creative portion only explicitly reached the Synthesis Level, as students learned to apply Design Patterns during project development, but did not explicitly require students to "make and defend judgements" as is required to demonstrate learning at the Evaluation Level.

3 IMPLEMENTING ATHLETIC SOFTWARE ENGINEERING

Implementing the Athletic Software Engineering pedagogy involves satisfying five high-level requirements: focus on skill mastery; provide training problems with video solutions; provide supervised and unsupervised practice, test, and incentivize competency; and build new skills from old ones. The following sections describe these requirements in more detail and provide examples.

1. Structure the curriculum as a sequence of skills to be mastered, not as concepts to be memorized.

While it can be useful, for example, for students to understand and be able to articulate the conceptual difference between black box and white box testing, athletic software engineering does not focus on developing that kind of capability. Instead, the pedagogy focuses on concrete skills (i.e., mechanics) whose acquisition can be demonstrated via simple problems not requiring a "creative leap" for their solution. In the case of this example, athletic software engineering would focus on the "mechanics" of developing a specified black (or white) box test given a specific tech stack. The mechanics might involve: cloning a repo using GitHub Desktop, setting up a project in IntelliJ IDEA for the local copy of the repo, creating a branch in git to hold the work associated with development of the test, writing the black (or white) box test using Mocha and Chai constructs, running and debugging the test using the command line, committing and pushing the branch, and finally merging the branch back into master.

Even if the test to be developed is specified at a high level (eliminating the need for a "creative leap"), then the mechanics of implementing the test case can still be nontrivial and involve an interplay between at least a half a dozen tech stack components and technologies.

Appendix A describes a sample syllabus from a recent semester of our software engineering course that describes the sequence of modules, the skill (mechanics) to be developed in each module, and the tech stack components required to exhibit the skill. All but one of the Athletic Software Engineering modules are 1 week long, the exception being one module that is 2 weeks in duration.

2. Design a set of "practice WODs" for each skill, each accompanied by a video that demonstrates their solution in "Rx" time.

Once the skill to be acquired through a module is identified, the instructor will often provide background readings about the skill, but it is required that the instructor provide carefully designed practice problems whose resolution helps develop competency with the skill. In addition, the instructor must also record an online video that shows the "reference solution" for the problem. The video solution time becomes the operational definition of "minimal" or "expert" time for problem solution.

In our class, we use CrossFit terminology instead of traditional school terminology to help reinforce the "athletic" nature of learning. So, instead of referring to "home assignments" or "in-class quizzes," we call them practice or in-class "Workout of the Day" ("WODs"). When referring to the times associated with training problems, we use "Rx" ("as prescribed") to refer to the expert time-to-solution. In addition to Rx time, we also provide an "Av" (advanced) and "Sd" (scaled) time limits, which are also acceptable solution times. Finally, each training problem specifies a "Did Not Finish" ("DNF") time, which represents the maximal amount of time allowed to solve the problem. DNF times are general 2×-3× the Rx time. DNF times typically range from 15 to 40 minutes. Students are required to learn how to solve WODs in less than the DNF time.

Athletic Software Engineering is the only software engineering pedagogy we know of in which a "maximum time for solution" is a first-class component of the pedagogy.

Appendix B provides an example of a practice WOD assignment.

3. Provide time for students to learn to solve the practice WODs in Rx time.

Once the students have access to the background readings and the practice WODs, they are assigned to attempt to solve each practice WOD until they can solve it in close to Rx time. If a student is working on a practice WOD and reaches DNF time, then he or she is told to stop, watch the solution video to diagnose their problem(s) with the mechanics, and then start over on the problem and see if the new time to solution is acceptable.

The number of practice WODs varies from module to module, but for our tech stack, the number varies from two to six practice WODs. The module with six practice WODs stretches over 2 weeks, as six practice WODs indicates too much skill building for a single week.

Athletic Software Engineering is the only software engineering pedagogy that we know of in which repeating homework assignments is a recommended practice.

4. Provide an in-class, group practice WOD so students can self-assess their competency with the skill.

Once the students have had a few days to train individually on practice WODs, the following class period is designed around a new practice WOD that students work together in small groups of two to three to solve. The goal of this session is to help the students to self-assess their competency with the skill, and to identify knowledge gaps and share insights with each other and through interaction with the instructor.

The group practice WOD is not graded, but it is timed and has a difficulty level similar to that of the in-class final WOD. Students leave the classroom with a better sense of their competency with the skill and are hopefully incentivized to work more on practice WODs if they are still not yet competent.

In recent semesters, we have started augmenting the in-class, group practice WOD with an optional, outside of class practice WOD that is administered by the Teaching Assistant (TA) for the class. The TA is responsible for designing the problem, administering it, and showing the students the solution. This provides an additional, valuable opportunity for students to self-assess whether they are ready for the in-class “final” WOD.

5. Test competency with the skill through an in-class final WOD that is graded on an all-or-nothing basis.

At the end of the modules, students are graded on their ability to individually solve a new problem correctly and prior to the stated DNF time. Appendix C provides an example of an in-class WOD assessment.

The in-class, final WOD is typically graded on an all-or-nothing basis: If the student solves it both correctly and within the time limit, then they get 100% credit; otherwise, they get no credit. This stringent grading standard for the final in-class WOD is deliberate, controversial, and important: We tell the students that if they follow the pedagogy, then they will accumulate enough practice to solve the final problem. If they do not follow the pedagogy, then they are taking a risk, because lack of competency will lead them to DNF and not receive credit.

We implement this all-or-nothing grading policy not to abuse students but to help them practice the pedagogy and complete the course successfully. Because skills often build upon each other during the semester, it is extremely important for students to acquire basic competency with each week’s skill to be set up to acquire competency with the following week’s skill.

In addition, the all-or-nothing grading policy incentivizes students to commit to the pedagogy of doing practice WODs, potentially more than once, to ingrain the mechanics to the point that they can complete the practice problems in approximately Rx time. Allowing partial credit effectively eliminates time as a constraint: It tells students that they do not really have to learn the skill well enough to apply it fluently, since they can get some credit just for making “some” progress on the problem during the assigned time. The all-or-nothing grading policy communicates to students that we expect them to receive full credit as long as they fully commit to the pedagogy.

The all-or-nothing pedagogy also places a responsibility on the instructor as well as the student: The instructor must design the practice WODs in such as way that, as long as a student can practice them enough to achieve Rx time, then they should be virtually guaranteed of passing the inclass final WOD for a module.

In our experience, some students ignore the pedagogy at the start of the semester, and then DNF on the WOD and receive no credit for that module. This serves as a wake up call, and they begin completing all the practice problems and attending the TA sessions, and then start to receive full credit. This is a natural and expected situation, we find that those students’ final grades do not suffer from this learning curve.

A substantial number of students DNF on the in-class WOD at the beginning of the semester for quite a different reason: they simply “freeze up” under the stress of the time limit and the all-or-nothing grading scheme. We tell those students that this is also a natural and expected situation. In fact, we explain that the in-class final WOD environment is quite similar to a software engineering interview environment, where the candidate is often given a problem and told to solve it on a white board in a few minutes time. The good news, we tell them, is that after 10 weeks of exposure to the stress of a timed, in-class WOD, they will lose the ability to “freeze up,” since they will have biologically habituated to this form of stress [37]. And that, furthermore, prior students have confirmed that they felt less stress in an interview environment because they had experienced WODs.

The result of all of this is a pattern that we see every single semester: for the first WOD of the semster, at least a third of the students DNF. By the final WOD of the semester, there has always been at least one WOD in which none of the students DNF'd.

The time-constrained nature of the inclass final WOD for a module has raised concerns for students with disabilities who normally request and receive accomodations in the form of extended time for test taking and separate testing rooms. This has been an issue for 1-2 students every semester. So far, we've been able to accommodate those students within the Athletic Software Engineering pedagogy successfully by explaining the motivation for it and asking them to try it temporarily.

Move on to the next skill, which typically builds upon competency with the prior skill.

Our course is structured such that each module (except for one) lasts for 1 week, and each module builds upon and/or reinforces the skills acquired during one or more prior modules. See Appendix A for a table showing the cumulative growth of skills assessed during an inclass final WOD over the course of the athletic portion of the course.

4 EVALUATION

4.1 Method

The goal of our evaluation process is to gather evidence regarding the following research questions:

- Is Athletic Software Engineering an effective pedagogy for learning software engineering?
- Do students comply with the basic components of Athletic Software Engineering?
- Does Athletic Software Engineering have side effects, such as creating competition, improving confidence, managing pressure, and improving focus?
- How can Athletic Software Engineering be improved?

To conduct this evaluation, we employed a mixed methods experimental design. We gathered and analyzed both self-reported, subjective questionnaire data at two points during the semester, as well as objective outcome data from in-class and at-home assignments. We gathered data from these four sources for a total of six semesters and a total of 286 students over the course of 4 years. For each research question, we analyzed multiple data sources with the goal of finding multiple, consistent sources of evidence for our conclusions.

Our four data sources are as follows:

(1) *Mid-semester questionnaire.* Each semester of the study period, soon after the Athletic Software Engineering portion of the course concluded, we administered a 12-item questionnaire that students completed anonymously. This questionnaire, presented in Appendix G, provides evidence regarding student attitudes about various aspects of the pedagogy. Two hundred fifty-six of 286 students (90%) completed the questionnaire.

(2) *WOD Card data.* As described in Appendix D, we use index cards for each student to record the time associated with each WOD and whether the student successfully completed it. Appendix E presents summary statistics for the success rates across WODs and students, as well as t-test results to test whether student performance improved over the course of the semester. We have WOD card data for 259 (91%) of the students; the remaining cards were lost or illegible.

(3) *Practice WOD repetition data.* On certain practice WOD assignments we asked students to self-report how many attempts they made at the practice WOD. These data help evaluate

whether this component of the pedagogy was being performed in practice. We have WOD repetition data for 230 (80%) of the students.

(4) *Course evaluation data.* Our institution requires the gathering of course evaluations through an online system that is made available to students during the last week of each semester. This data are released to instructors after the semester ends. Post course evaluation data provide evidence regarding student attitudes toward the entire course, which included both an athletic component and a project component. We have course evaluation data for 212 (74%) of the students. Course evaluation data are anonymous.

The students whose data were used in this study are almost all computer science and computer engineering undergraduates at the University of Hawaii. (One or two students a year might be graduate students fulfilling undergrad requirements, or students from another degree program or university.) All of them had a minimum of two prior computer science or computer engineering programming courses in either C++ or Java. Some students had additional prior coursework or experience in networking, graphic design, databases, or application development. We did not attempt to collect demographic data on computer science background, and since much of the data was collected anonymously, we could not have correlated it to other responses anyway.

In this evaluation, we do not perform a longitudinal analysis; i.e., we do not, for example, claim that there are trends by comparing data from fall 2015 to fall 2018. This is for a variety of reasons: preliminary analysis did not discern any meaningful trends over the course of the 4 years, and changes in the course content (such as changes to elements of the tech stack) form a significant confounding variable affecting the validity of any conclusions from these comparisons. Instead, our evaluation procedure combines the data from all 4 years, although we will provide breakouts for individual years when that appears useful to understanding the data.

4.2 Is Athletic Software Engineering an Effective Pedagogy for Learning Software Engineering?

There is strong evidence to suggest that Athletic Software Engineering is an effective pedagogy. Responses to Question 1 in the questionnaire indicates that a majority of respondents each semester prefer it to traditional approaches to teaching computer science and programming (i.e., "longer assignments and problems, no time limit, no instructor solution videos"). Over the six semesters, the preference for Athletic Software Engineering ranges from a low of 82% to a high of 94%, with an aggregate average of 88% preference for Athletic Software Engineering. Of the remaining 12% of respondents, 7% recommended a hybrid approach, leaving 5% with an absolute preference for a traditional approach.

We analyzed the free text response asking why students answered this way and found three recurrent themes. First, students felt that they learned more from this pedagogy (43% of respondents). One student wrote, "I think we covered a lot more material than would be practical for the traditional approach," while another said, "Athletic Software Engineering forced us to learn the content of each module to the point where we were more than proficient with it to complete the WOD in RX/AV time. Traditional assignments don't force you to learn the content to that extent." Second, students felt that the pedagogy incentivized them to study and to keep up with the material (22%). One student wrote, "I like having strict time limits on assignments to force myself to focus on the assignment, instead of multi-tasking other things while working on the assignment." Third, students felt that the pedagogy helped them acclimate to the pressures associated with "real world" software development (15%). One student wrote, "I got accepted to an internship from Amazon and they put me through 3 rounds of 'WODs' to test me before giving me the offer. I think the WODs in 314 helped."

Most of the students who preferred the traditional approach disliked the time constraints. One student wrote: "I feel like the timing of the assignments for me doesn't have a positive effect on my end product. Sometimes I feel like I learn the content for speed instead of spending the time to actually learn the purpose of the content. This is not a fault of the system, this is more a personal fault of not repeating WODs or revisiting old assignments." Another student wrote, "I understand this approach and how it's more of a hands on approach but for me I have a hard time working under any time constraint and it makes me very nervous."

Another source of evidence regarding the effectiveness of Athletic Software Engineering can be found in the Course Evaluation data, as summarized in Appendix H. These data indicate that over 90% of the students "agreed" or "strongly agreed" that they "gained a good understanding of concepts/principles in this field." However, since the course had both Athletic and Project-based components, these results provide only weak evidence by themselves (it is possible that the students could have gained good understanding only through the project-based component of the class). However, these results certainly do not provide contradictory evidence to the claim that Athletic Software Engineering is an effective pedagogy.

We believe that the data provides evidence that Athletic Software Engineering is an effective pedagogy. It does not, however, provide evidence that Athletic Software Engineering is a *superior* pedagogy. To test that claim, it would be necessary to compare outcome data (say, final project quality) from a class taught using Athletic Software Engineering to a class covering the same material but taught using some other pedagogy. This limitation and others are discussed in Section 5.

4.3 Do Students Comply with the Basic Components of Athletic Software Engineering?

Several questions in the Questionnaire were designed to determine the level of compliance with the components of the Athletic Software Engineering pedagogy.

First, we asked students if repeating the practice WODs was useful. Eight-four percent of the respondents said that they found that repeating the practice WODs was useful for at least some of the modules. Most students said that repeating the practice WODs helped them learn more. One student wrote, "Repeating them were definitely useful because during each try, I realized there were different things I forgot/didn't really know. I ran into different problems and after a few repeats I was able to fully understand the whole problem. Repeating it also helped me to remember certain functions and how things worked." Another wrote, "Yes! We learn one approach by trying it ourselves. Then we learn another approach watching the screencast. Then we implement this second approach by repeating the WOD again." Finally, one student wrote, "There were things that I didn't quite understand until I re-did the WOD for the 3rd or 4th time."

The 16% of respondents who did not find repeating the WODs to be useful generally did not think that they would encounter new issues, or that it would be worth the investment in time. One student wrote, "No, I did not find repeating the practice WODs to be extra useful. Completing them only once was more than enough for me, and repeating them (especially after knowing the solution) didn't help with my retention, since it's the exact same problem." Another wrote, "I rarely repeated the WODs to be honest. With my full coursework I did what was required." Another wrote, "I never repeated any WODs... When doing the same WOD repeatedly, people tend to start memorizing instead of learning." What is interesting about these responses is that many of the negative responses seem based on an *a priori* conclusion that repeating the practice WOD would not be helpful, rather than "suspending disbelief" and finding out through direct experience.

To get a sense for the amount of repetition of practice WODs, we asked students to self-report how many times they repeated certain practice WODs. The results are summarized in Appendix F. The data shows that 65% of the students repeated the WOD, and supports the questionnaire data indicating that the majority of students repeat the practice WODs. It also shows that, for this sampling of practice WODs, approximately a third of the students repeated the WOD three or more times.

Second, we asked students if the in-class, graded, all-or-nothing final WOD for a module was useful. Ninety-one of the respondents said it was useful, indicating that it incentivized them to learn the material and keep up with the class. One student wrote, “This made sure that I actually did the practice WODs beforehand, otherwise, I probably would not have paid as much attention to the practice WODs.” Another wrote, “Yes, the grade definitely put an emphasis on the importance of learning and understanding the material. If I ever fell behind, I knew the following WODs would be much more challenging. The fact that they were graded also pressured me into grasping every concept and being able to apply them effortlessly.” Another student wrote, “I believe having an in-class graded WOD was helpful because it forced me to know the material before coming into class, rather than just showing up to class. I felt that having them held me accountable for my learning.” Another said, “I think it was helpful because it was 100 or 0. It gave me extra motivation to find the solution and made sure it worked. If we were to receive partial credit and such, then I believe I would of settled for partial points at times.” Similarly, another student wrote, “If there was no in-class WOD worth 100 pts, I would have shown up for far fewer classes.”

The 9% of respondents who did not find the in-class graded WODs to be useful often singled out the grading scheme and time pressure. One student wrote, “The assignments were only worth 10 each while the WODs were 100. It was stressful knowing your grade would suffer tremendously if you screwed up on the WODs.” Another wrote, “I do not think it was helpful, as I have stated before I do not work well under a certain time constraint and it stresses me out a lot.”

Third, to evaluate the performance component of the pedagogy, one question asked students if they tried to improve their performance (from DNF to Sd, Sd to Av, Av to Rx) over the course of the semester. Only 55% of students said they tried to improve their performance. One student wrote, “I did try to improve my performance over the course of the semester. Doing the WODs within Av or Rx shows I know and that I’m familiar with the content.” Most of the remaining half said that their only concern was to avoid a DNF time, since the grading scheme did not differentiate between Rx, Av, and Sd. Another student wrote, “No, even when I finish, I ensure that my code works completely. A working Sd is better than a Rx with a careless mistake.” Another wrote, “Since it was either pass or fail, I didn’t really try to improve, I just strived to not get a DNF.”

Although only about half of the students said they attempted to improve their performance, among the remaining responses, no one actually indicated that the performance indicators were bad. Instead, they said they simply ignored them. Based on this, we view performance indicators as a positive component of the pedagogy for about half of the students and a benign component for the remaining half.

In summary, the data suggest that students find the principle components of athletic software engineering to be useful: the use of practice WODs, of solution videos, of timed exercises, and even the admittedly stress-inducing in-class WOD with an all-or-nothing grading scheme. Many students indicate that they see how these components fit together to create an atypically productive learning environment.

4.4 Does Athletic Software Engineering Have Side Effects, Such as Creating Competition, Improving Confidence, Managing Pressure, and Improving Focus?

A few questions on the Questionnaire looked for side-effects of the pedagogy. One question asked if the WOD format produced competitive feelings, such as wanting to finish earlier than others. Sixty-seven percent of respondents answered that it produced competitive feelings. We also asked if that was a positive or negative aspect of the pedagogy. Sixty percent of the students said it was positive, 9% said it was negative, and the remainder found this issue to be neither positive nor negative. One representative comment was, “Competition is always good when learning cause it lights a fire within you to try better. It sets a goal that you want to reach and the feeling of reaching that goal is awesome.”

Although few students viewed the competitive side-effect as negative, one student’s response is worrisome: “Actually having people rush to finish first was discouraging for me. I feel that CS in general has a very competitive culture. If you are not fast, knowledgable, or been coding before college, you suck.” Given the current problems with diversity and retention in computer science [1, 19], this comment indicates the possibility that Athletic Software Engineering might have a disproportionate, negative impact on women and underrepresented groups. Unfortunately, because the enrollment of women was so low in the courses analyzed for this study, and because so much of the data was collected anonymously, it was not possible to perform analyses for gender-related effects. We do not have data on the number of students from underrepresented groups, but it would be similarly difficult to conduct analyses based upon background.

A second question asked if WODs helped increase confidence in their software development capabilities. Eight-four percent of students said it did. One student commented, “Yes, I feel more confident in my software development abilities. Not necessarily from the WODs themselves, but from the practice and assignments leading up to the WODs. It’s not about the destination, it’s about the journey. Doing a practice WOD 3 times the night before is definitely instrumental in becoming sharper in software skills, be it git, html, javascript, etc.”

A third question asked if WODs helped students get used to “programming under pressure.” Seventy-nine percent of students said it did. One student commented, “Yes, I did become more comfortable programming under pressure. After constantly being under a time constraint, the nerves slowly went away and I learned to focus more.” Another student said, “If the first time you program under pressure is in a coding interview, that is going to be a bad day, so I’m thankful for the experience.”

Finally, a question asked if the structure of the practice WODs helped students to improve their focus and concentration while doing homework. Eighty-two percent of the students indicated that it did. One student commented, “The being-timed aspect made me focus on the one task. I would ignore my phone and other people while I worked.” Another student noted the utility of the solution video for aiding focus: “Yes, I feel the homework WOD helped me focus more while doing them. By having the screen cast there as an aid you know that if you get stuck somewhere you have something to help you. So you can give the WOD your best and if you get stuck the answer is there to help and you would be able to practice the WOD again.” Another student commented, “Doing homework seemed like less of a chore and more of a competitive challenge. It was easier to gain motivation for the homework because I could know that I would be finished with my homework within a rough time period. I could set aside double the DNF time for homework and know that I will take no more than that amount of time to complete the assignment.”

In summary, Athletic Software Engineering has a number of side-effects beyond just the acquisition of skills involving competition, confidence, pressure, and focus. In all of these cases, the majority of students found these side-effects to be positive, although we are concerned about the possibility of competition having negative implications on women and underrepresented groups.

4.5 How can Athletic Software Engineering Be Improved?

One section of the questionnaire asked students how they would change the class to make it more effective for future students. Analysis of the responses indicate that approximtely a third of the students recommend changes to the *content* of the course. For example, some students wished for different components in the technical stack, others for additional content of databases or html, or increased educational resources such as "cheat sheets" for various technologies. We agree emphatically that the course content can be improved, and each semester we introduce changes and additions to the content.

With respect to the Athletic Software Engineering pedagogy, 9% of the students advocated for a change to the grading scheme for WODs, which typically involved introducing partial credit to reduce the stress and anxiety associated with them. While we understand that it is a natural desire to reduce stress and anxiety, we are less sure that this is ultimately in the interests of most students. The other data collected indicates that the all-or-nothing nature is an incentive to keeping up. In addition, most students actually gain something valuable from the experience of the pressure.

Last, 13% of the students advocated for no change at all to the course: They like it just the way it is.

5 LIMITATIONS

There are a variety of limitations with respect to the internal and external validity of this study.

First, an alternative, and potentially more rigorous experimental design would have split the student population into two groups, one of which would have experienced athletic software engineering, the other of which would have experienced a "control" pedagogy. Such a design can yield evidence for the performance impact of explicit training in the "mechanics" of the tech stack. Although this is an interesting path for future research, we did not pursue this design in the current research for two reasons. First, as noted in Section 1, there are a wide variety of pedagogies for software engineering education, so it is not clear what to choose as the "control" pedagogy. Perhaps more importantly, our surveys showed a consistent, overwhelming preference for the athletic software engineering pedagogy. At least for this study period, we felt that student preference should be prioritized over experimental design rigor.

Second, we chose not do a longitudinal study in which we compared current final project quality to the quality of projects created prior to the introduction of athletic software engineering. While we believe anecdotally that the projects have improved in quality, the tech stack has also changed significantly in the past 5 years. It would be difficult to disentangle the impact of the change in tech stack from the impact of the change in pedagogy.

Third, interpretation of these results are limited by the self-reported nature of the data. Students could have provided positively biased results for some reason besides the pedagogy. We made efforts to minimize this risk. The surveys were anonymous, and we told the students that their opinions, positive or negative, would not influence their grades, but would rather be useful in improving the course in future.

Fourth, our design does not allow us to see if student evaluations of Athletic Software Engineering were influenced by their adherence to the pedagogy. For example, is there a relationship between DNF-ing a WOD and not training correctly? Did students who had a negative perception of Athletic Software Engineering practice the technique correctly? Unfortunately, due to the self-reported nature of the data, as well as the anonymous nature of the questionnaire, it is not possible for us to investigate these research questions within the current study.

Fifth, the external validity, or generality of these findings are limited by the fact that they were collected within a single type of software engineering class taught by a single instructor. Determining the external validity of this research requires additional studies by other instructors in other organizations to gain insight into the impact of variations in instructional styles, variations in student demographics, or variations in course content.

Sixth, our classroom size was small, varying between 15 and 30 students. We discuss the issues involved with scaling the size of the classroom in Section 6.

6 CONCLUSIONS AND FUTURE DIRECTIONS

This research presents a new pedagogy called Athletic Software Engineering, a well-structured pedagogy based heavily on repeated practice assignments focusing on developing fluency with the "mechanics" of technology usage. Progress is assessed using time-controlled exercises with specified time requirements, inspired by athletic workouts, to overcome the challenges with teaching students full stack development. The research includes four types of data collection, gathered over 4 years and six semesters involving a total of 286 students. Results indicate that Athletic Software Engineering is an effective approach to software engineering education. We also discuss the limitations of this research design. Our recommendations for future directions are mostly directed toward addressing these limitations.

First, this research was based upon one software engineering class taught by a single instructor. This approach was useful for "controlling" certain confounding variables in the results, but also limits the external validity of the findings. It would be helpful to see how the pedagogy works in other software development settings including different material and different instructors. There is preliminary data regarding this, but more work is needed to better understand the generality of the approach [18, 20].

Second, while the current research design is able to provide evidence supporting the claim that Athletic Software Engineering is an effective pedagogy, it is not capable of providing rigorous evidence indicating that Athletic Software Engineering is superior to any other pedagogy. (There is some anecdotal evidence in the form of student comments that the pedagogy helped them learn more than they would have in a "traditional" setting). It would be useful to create an experimental design in which different students are randomly assigned to different pedagogies, and that outcome measures (such as final projects) are gathered to see if there are observable differences.

Third, while our course design involved Athletic Software Engineering for the first half of the course, and project-based pedagogy for the second half, other designs are possible. For example, one could interleave the Athletic and non-Athletic by alternating between them on a weekly basis.

Fourth, the results indicate that a significant number of students felt that Athletic Software Engineering produced competitive feelings within them. While this could be a positive motivating force, some research suggests that this could be disproportionately de-motivating for women and underrepresented groups. It would be useful to investigate this issue further, and to see if there is a way to retain the benefits of the Athletic Software Engineering pedagogy while mitigating the downsides of competition.

Fifth, we believe that Athletic Software Engineering has the potential to be scaled to large teaching environments such as Massively Open Online Courses (MOOCs). Scaling the approach up requires ways to assess performance times without in-class observation, and ways to support group work without co-location. If these structural problems can be addressed, then a MOOC environment provides an interesting environment to introduce game mechanics (i.e., leaderboards) as well as the gathering of additional demographic data to better understand the strengths and limitations of this approach.

Sixth, while our experience to date with Athletic Software Engineering incorporates aspects of project-based and flipped classroom pedagogies, we do not have any experience with studio-based learning. It would be interesting to see how this approach might be adapted to studio-based learning, and if new insights about either pedagogy emerge from the experience. For example, it seems likely that incorporation of studio-based learning techniques would lead to an explicit focus on the "top" of Bloom's Taxonomy, the Analysis Level, as students in a studio would likely be required to "make and defend judgements based on internal evidence or external criteria."

APPENDICES

A SAMPLE SYLLABUS

The following table shows a sample syllabus for the "athletic" portion of our introductory software engineering class, along with the skills to be developed in each module. The first 10 weeks of the course are devoted to the athletic software engineering pedagogy. The last 6 weeks are organized around a "project" pedagogy, although there is typically an athletic module on Testing at some point after the 11th week.

"DNF Time" is the maximum amount of time allowed for students to complete a problem assessing competency with the skill and tech stack at the end of the week.

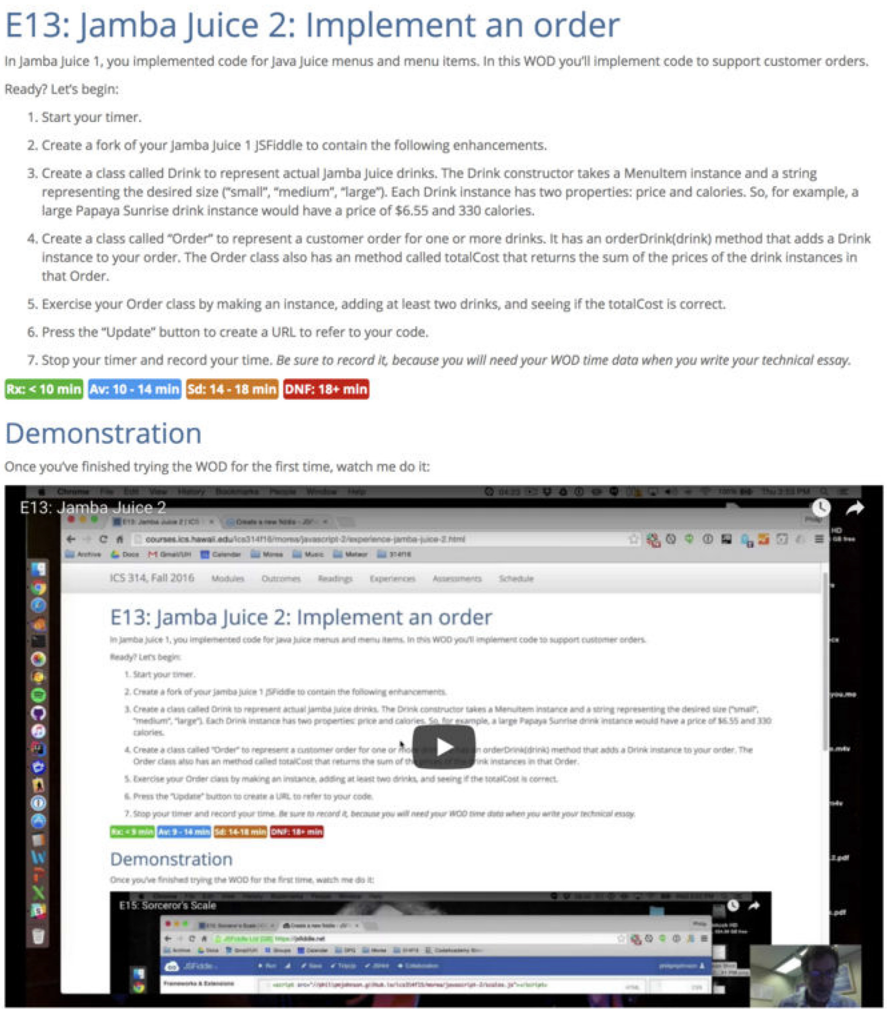
Note that by the 8th week, the students must manipulate a tech stack containing 10 technologies to complete the WOD. Also note that the technology "JSFiddle" is used as a bootstrapping mechanism and thus is part of the tech stack only for the first 3 weeks, after which it is discarded.

Table 3. Sample Syllabus

Week	**Module(s)**	**Skill, Tech Stack, DNF Time**
1	Basic Javascript	*Skill*: basic competency with Javascript syntax by solving a problem with a simple Javascript function. *Tech Stack*: JSFiddle, Javascript. *DNF time*: 10 minutes
2	OO Javascript	*Skill*: basic competency with object oriented Javascript constructs by solving a problem requiring design of two Javascript classes. *Tech Stack*: JSFiddle, Javascript. *DNF time*: 20 minutes.
3	Functional Javascript	*Skill*: basic competency with the Underscore library by solving a problem requiring use of functional constructs such as reduce(), map(), filter(), each(), and so on. *Tech Stack*: Javascript, JSFiddle, Underscore. *DNF time*: 30 minutes.
4	IDE, CM, Coding Standards	*Skill*: basic competency with writing Javascript using IntelliJ IDEA that conforms to coding standards expressed with ESLint. *Tech Stack*: Javascript, Underscore, IntelliJ IDEA, git, GitHub, ESLint. *DNF time*: 20 minutes.
5	HTML, CSS	*Skill*: basic competency with HTML and CSS. *Tech Stack*: IntelliJ IDEA, git, GitHub, HTML, CSS *DNF time*: 20 minutes
6	CSS Frameworks	*Skill*: basic competency with a CSS framework (Semantic UI). *Tech Stack*: IntelliJ IDEA, git, GitHub, HTML, CSS, Semantic UI. *DNF time*: 45 minutes
7	React	*Skill*: basic competency with the React UI framework. *Tech Stack*: Javascript, IntelliJ IDEA, git, GitHub, HTML, CSS, Semantic UI, React. *DNF time*: 50 minutes
8–10	Meteor, Mongo	*Skill*: basic competency with Meteor application framework and Mongo database system. *Tech Stack*: Javascript, IntelliJ IDEA, git, GitHub, HTML, CSS, Semantic UI, React, Meteor, Mongo. *DNF time*: 30 minutes
11+	Testing	*Skill*: basic competency with writing unit tests. *Tech Stack*: Javascript, IntelliJ IDEA, git, GitHub, mocha, chai. *DNF time*: 15 minutes.

B SAMPLE PRACTICE WOD

Here is an example of a practice WOD from the object oriented Javascript module. The video embedded in the page is available at https://www.youtube.com/watch?v=xS1uVqIG9hE.



E13: Jamba Juice 2: Implement an order

In Jamba Juice 1, you implemented code for Java Juice menus and menu items. In this WOD you'll implement code to support customer orders.

Ready? Let's begin:

1. Start your timer.
2. Create a fork of your Jamba Juice 1 JSFiddle to contain the following enhancements.
3. Create a class called Drink to represent actual Jamba Juice drinks. The Drink constructor takes a MenuItem instance and a string representing the desired size ("small", "medium", "large"). Each Drink instance has two properties: price and calories. So, for example, a large Papaya Sunrise drink instance would have a price of $6.55 and 330 calories.
4. Create a class called "Order" to represent a customer order for one or more drinks. It has an orderDrink(drink) method that adds a Drink instance to your order. The Order class also has an method called totalCost that returns the sum of the prices of the drink instances in that Order.
5. Exercise your Order class by making an instance, adding at least two drinks, and seeing if the totalCost is correct.
6. Press the "Update" button to create a URL to refer to your code.
7. Stop your timer and record your time. *Be sure to record it, because you will need your WOD time data when you write your technical essay.*

Rx: < 10 min Av: 10 - 14 min Sd: 14 - 18 min DNF: 18+ min

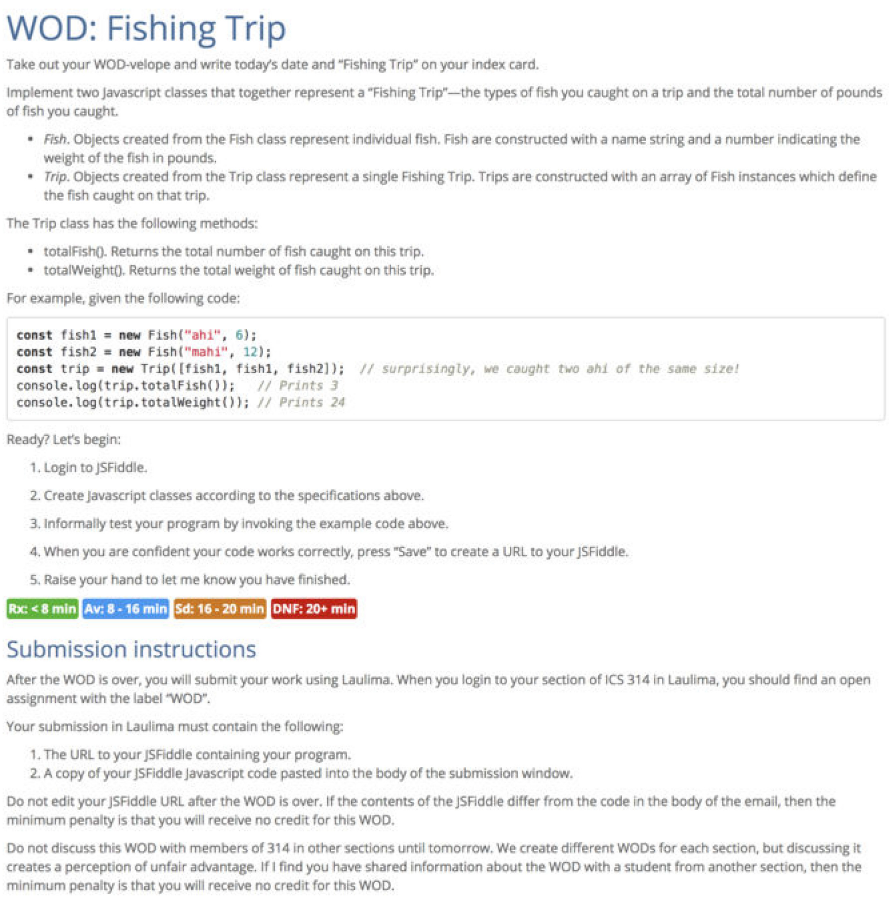
Demonstration

Once you've finished trying the WOD for the first time, watch me do it:

Fig. 1. An example practice WOD from the Object Oriented Javascript module.

C SAMPLE IN-CLASS WOD

Here is an example of an in-class WOD from the object oriented Javascript module.



WOD: Fishing Trip

Take out your WOD-velope and write today's date and "Fishing Trip" on your index card.

Implement two Javascript classes that together represent a "Fishing Trip"—the types of fish you caught on a trip and the total number of pounds of fish you caught.

- *Fish*. Objects created from the Fish class represent individual fish. Fish are constructed with a name string and a number indicating the weight of the fish in pounds.
- *Trip*. Objects created from the Trip class represent a single Fishing Trip. Trips are constructed with an array of Fish instances which define the fish caught on that trip.

The Trip class has the following methods:

- totalFish(). Returns the total number of fish caught on this trip.
- totalWeight(). Returns the total weight of fish caught on this trip.

For example, given the following code:

```
const fish1 = new Fish("ahi", 6);
const fish2 = new Fish("mahi", 12);
const trip = new Trip([fish1, fish1, fish2]);  // surprisingly, we caught two ahi of the same size!
console.log(trip.totalFish());    // Prints 3
console.log(trip.totalWeight()); // Prints 24
```

Ready? Let's begin:

1. Login to JSFiddle.
2. Create Javascript classes according to the specifications above.
3. Informally test your program by invoking the example code above.
4. When you are confident your code works correctly, press "Save" to create a URL to your JSFiddle.
5. Raise your hand to let me know you have finished.

Rx: < 8 min Av: 8 - 16 min Sd: 16 - 20 min DNF: 20+ min

Submission instructions

After the WOD is over, you will submit your work using Laulima. When you login to your section of ICS 314 in Laulima, you should find an open assignment with the label "WOD".

Your submission in Laulima must contain the following:

1. The URL to your JSFiddle containing your program.
2. A copy of your JSFiddle Javascript code pasted into the body of the submission window.

Do not edit your JSFiddle URL after the WOD is over. If the contents of the JSFiddle differ from the code in the body of the email, then the minimum penalty is that you will receive no credit for this WOD.

Do not discuss this WOD with members of 314 in other sections until tomorrow. We create different WODs for each section, but discussing it creates a perception of unfair advantage. If I find you have shared information about the WOD with a student from another section, then the minimum penalty is that you will receive no credit for this WOD.

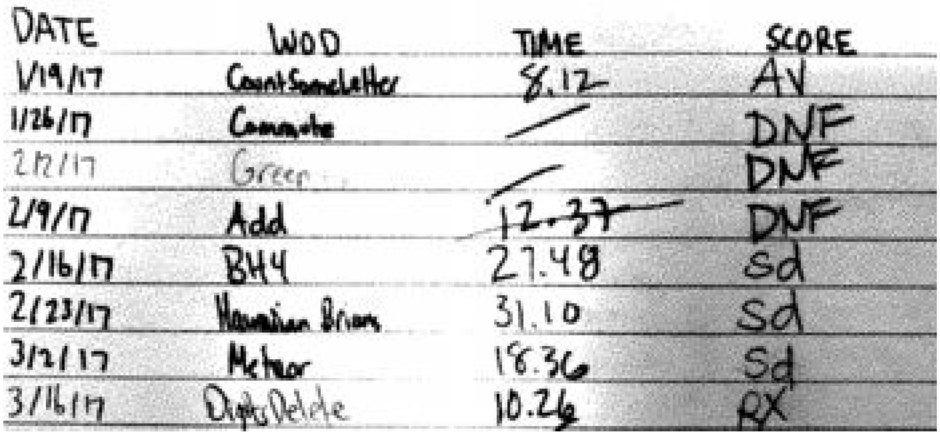
Fig. 2. An example in-class WOD from the Object Oriented Javascript module.

D RECORDING IN-CLASS WOD DATA EFFECTIVELY

To ensure that in-class WODs are recorded effectively, we keep a set of envelopes, one per student, with their name on it and an index card inside that records each attempt at an in-class final WOD for a module. At the beginning of the class period, they fill out the date and the name of the WOD. The instructor keeps a running time during the WOD, and when the student completes it, he or she raises his or her hand. The instructor then writes down the time associated with the WOD.

Once the WOD is over, the students go through a process to submit their work through an online assignment portal. After class, the instructor or TA checks the submission. The student receives credit for the WOD only if they completed the assignment within the time limit, and the submission solved the problem correctly.

This card illustrates two features of the Athletic Software Engineering pedagogy. First, you can see that in the first half of the course, the student DNF'd three of four times. But in the second half, the student successfully completed the WOD every time.



DATE	WOD	TIME	SCORE
1/19/17	CountSameLetter	8.12	AV
1/26/17	Commute	/	DNF
2/2/17	Green	/	DNF
2/9/17	Add	~~12.37~~	DNF
2/16/17	BH4	27.48	Sd
2/23/17	Hawaiian Brian	31.10	Sd
3/2/17	Meteor	18.36	Sd
3/16/17	DigitsDelete	10.26	RX

Fig. 3. An example index card containing WOD data.

Second, the card shows that on 2.9/17, the student thought that they had completed the WOD successfully after 12 minutes and 37s. Unfortunately, when the submission was later reviewed, it was found to be incorrect, so the time was crossed out and the WOD was scored as a DNF.

E IN-CLASS WOD SUMMARY STATISTICS

This section summarizes data for eight in-class WODs performed by a total of 258 students for the study period (fall 2015 to fall 2018), for a total of 1,538 individual scores. Our analysis represents the outcome of a WOD as either success (Rx, Av, Sd) or failure (DNF). Since WODs vary from semester to semester, and since the timings for Rx, Av, and Sd are chosen somewhat arbitrarily, we do not feel a more fine-grained analysis (i.e., Rx vs. Sd) would be meaningful. But we do feel that summmary statistics on the pass-fail rate provide a sense for the challenge associated with the module-level evaluation.

First, let us look at the distribution of success on a per-student basis.

Table 4. Individual Success Rate: WODs Passed

8	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
26%	26%	12%	12%	8%	12%	4%	1%	0%

This table shows that every student passed at least one WOD during the semester. Over half (52%) either completed all of the WODs successfully or failed on only one WOD.

This next table shows the average success rate per WOD for the first eight WODs.

Table 5. Average Success Rate for First Eight WODs

WOD 1	**WOD 2**	**WOD 3**	**WOD 4**	**WOD 5**	**WOD 6**	**WOD 7**	**WOD 8**
68%	63%	55%	72%	91%	85%	79%	82%

On average, over 30% of the students fail the first three WODs, but the failure rate decreases to 10%–20% after that. We used a t-test to compare the scores on the first four WODs to the scores on the last four WODs and found that this difference is statistically significant.

Table 6. *t*-Test Comparison of Scores on First Four WODs vs. Last Four WODs

	WODs 1-4	WODs 5-8
Mean	0.648543689320388	0.84548105
Variance	0.228156283	0.130769929
Observations	1030	1029
Hypothesized Mean Difference	0	
df	1917	
t stat	−10.54793064	
P(T<=t) one-tail	1.26374E-25	
t Critical one-tail	1.645648885	
P(T<=t) two-tail	2.52749E-25	
t Critical two-tail	1.961202244	

F PRACTICE WOD ATTEMPT RESULTS

On selected practice WODs, we ask students to self-report how many times they attempted the WOD when they turn in the assignment. We selected one such practice WOD and recorded student self-reported data for fall 2016 through fall 2018 (results were not available for fall 2015).

We were able to collect data from 230 students. The following table summarizes the number of attempts (from one attempt to eight attempts), the count of the number of students who attempted the practice WOD that many times (from 81 for one attempt to 1 person who attempted the same practice WOD eight times), and the percentage associated with that count.

Table 7. Number of Practice WOD Attempts

Attempts:	1	2	3	4	5	6	7	8
Count:	81	83	46	10	5	3	0	1
Percentage:	35%	36%	20%	4%	2%	3%	0%	0.4%

The results indicate that approximately two thirds of the students attempted this practice WOD more than once. Half of the students who repeated the practice WOD more than once repeated it three or more times.

G ATHLETIC SOFTWARE ENGINEERING QUESTIONNAIRE RESULTS

The following anonymous questionnaire was administered in fall 2015 (one section, 17 responses), fall 2016 (one section, 28 responses), spring 2017 (two sections, 56 responses), Fall 2017 (two sections, 57 responses), spring 2018 (two sections, 55 responses), and fall 2018 (two sections, 43 responses), for a total of six semesters and 256 students. During these semesters, a total of 286 students were enrolled, yielding a 90% response rate.

Summary statistics on the responses to each question are provided in *italics*.

1. Given the choice between learning this semester's material using athletic software engineering or learning it in a more traditional way (longer assignments and problems, no time limit, no instructor solution videos) which approach would you choose?

- I would choose athletic software engineering.
- I would choose a more traditional approach.

Eight-eight percent of respondents would choose athletic software engineering, and 12% would choose a more traditional approach.

2. Please explain your choice.

Of those preferring athletic software engineering, 42% indicated that it improved their learning, 20% indicated that it provided useful incentives and/or improved their self-discipline, 15% indicated that it helped them get used to pressure and/or "real world" situations, 5% indicated that it provided an active, engaging learning environment, and 12% indicated that the solution video was useful for learning.

This question also clarified that of the 12% who did not indicate a preference for athletic software engineering, and 7% advocated for some sort of mixed or hybrid approach. This leaves 5% of respondents who unilaterally preferred a traditional approach.

3. Did you find repeating the practice WODs to be useful? Why or why not?

Eighty-four percent of respondents found repeating the practice WODs to be useful.

4. Do you think that having an in-class, graded WOD was helpful for learning? Why or why not?

Ninety-one percent of respondents found that having an in-class graded WOD was helpful for learning.

5. Did you like being able to leave the room after finishing the in-class WOD? Why or why not?

Eighty percent of respondents liked being able to leave the room.

6. Did you try to improve your performance (from DNF to Sd, Sd to Av, and Av to Rx) over the course of the semester? Why or why not?

Fifty-five percent of respondents tried to improve their performance.

7. Did the WODs produce any competitive feelings within you, such that you wanted to finish earlier than some others in the class?

- Yes, the WODs produced some competitive feelings.
- No, the WODs did not produce any competitive feelings.

Sixty-seven percent of respondents felt that the WODs produced competitive feelings.

8. If the WODs produced competitive feelings, do you feel this was a positive aspect of the class? Why or why not?

Sixty percent of respondents felt that competitive feelings were positive, 9% felt they were negative, and the remainder had no feelings about this one way or the other.

9. Do you feel that doing WODs has helped you feel more confident about your software development capabilities? Why or why not?

Eighty-four percent of respondents felt that the WODs helped them feel more confident.

10. What is one thing you would change about the class to make it more effective for future students?

There were many answers to this question. Approximately 36% of the answers involved changes to the content of the course, not the pedagogy. Fourteen percent of the answers said there was nothing that should be changed. Approximately 9% wished for a change to the all-or-nothing grading policy on in-class WODs.

11. One goal for the in-class WODs is to help you get used to "programming under pressure." Do you feel you became more comfortable programming under pressure over the course of the semester? Why or why not?

Seventy-nine percent of respondents felt that the WODs helped them program under pressure.

12. One goal for the practice (homework) WODs is to improve your focus and concentration while doing homework. Did you feel that WOD-based homework assignments helped you to be more focused while doing homework? Why or why not?

Eight-two percent of respondents felt that the WODs helped them improve their focus.

H COURSE EVALUATION RESULTS

Our institution has an online system to gather course evaluation data that is available to students during the last week of the semester. These data contrast with the Athletic Software Engineering questionnaire, in that this data reflects student views regarding the entire course, which comprises both Athletic and Project-based pedagogies.

For this study, we present the data from a single question: "I gained a good understanding of concepts/principles in this field in Table 8." We believe this is most useful for providing evidence regarding the Athletic Software Engineering pedagogy.

Table 8. Responses to "I Gained a Good Understanding of Concepts/Principles in This Field"

Semester	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Fall 2015	0	0	0	7	9
Fall 2016	1	0	2	7	17
Spring 2017 (1)	0	0	3	4	21
Spring 2017 (2)	0	1	1	11	17
Fall 2017 (1)	0	1	4	11	11
Fall 2017 (1)	0	1	1	9	17
Spring 2018 (1)	0	0	2	10	16
Spring 2018 (2)	0	0	1	17	10
Totals	*1 (0.5%)*	*3 (1%)*	*14 (7%)*	*76 (36%)*	*118 (56%)*

REFERENCES

[1] Jerri Barrett. 2017. Expanding the Pipeline: Key Learnings on Retaining Underrepresented Minorities and Students with Disabilities in Computer Science. Retrieved from https://goo.gl/CQfy2K.

[2] Benjamin Bloom. 1956. *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain.* David McKay Co Inc., New York, NY.

[3] A. Bollin, E. Hochmüller, and R. T. Mittermeir. 2011. Teaching software project management using simulations. In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET'11).* 81–90. DOI: https://doi.org/10.1109/CSEET.2011.5876160

[4] Charles Bonwell and James Eison. 1991. *Active Learning: Creating Excitement in the Classroom.* ERIC Clearinghouse on Higher Education,.

[5] Tom Briggs. 2005. Techniques for active learning in CS courses. *J. Comput. Sci. Coll.* 21, 2 (Dec. 2005), 156–165. http://dl.acm.org/citation.cfm?id=1089053.1089075

[6] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. 2015. Software engineering project courses with industrial clients. *Trans. Comput. Educ.* 15, 4 (Dec. 2015), 17:1–17:31. DOI: https://doi.org/10.1145/2732155

[7] C. N. Bull and J. Whittle. 2014. Supporting reflective practice in software engineering education through a studio-based approach. *IEEE Softw.* 31, 4 (Jul. 2014), 44–50. DOI: https://doi.org/10.1109/MS.2014.52

[8] Chung-Yang Chen and P. Pete Chong. 2011. Software engineering education: A study on conducting collaborative senior project development. *J. Syst. Softw.* 84, 3 (Mar. 2011), 479–491. DOI: https://doi.org/10.1016/j.jss.2010.10.042

[9] David Coppit. 2006. Implementing large projects in software engineering courses. *Comput. Sci. Educ.* 16, 1 (Mar. 2006), 53–73. DOI: https://doi.org/10.1080/08993400600600443

[10] M. Daun, A. Salmon, T. Weyer, K. Pohl, and B. Tenbergen. 2016. Project-based learning with examples from industry in university courses: An experience report from an undergraduate requirements engineering course. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET'16).* 184–193. DOI: https://doi.org/10.1109/CSEET.2016.15

[11] Heidi J. C. Ellis, Gregory W. Hislop, Stoney Jackson, and Lori Postner. 2015. Team project experiences in humanitarian free and open source software (HFOSS). *Trans. Comput. Educ.* 15, 4 (Dec. 2015), 18:1–18:23. DOI: https://doi.org/10.1145/2684812

[12] Hakan Erdogmus and Cecile Peraire. 2017. Flipping a graduate-level software engineering foundations course. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track (ICSE-SEET'17).* IEEE Press, Los Alamitos, CA, 23–32. DOI: https://doi.org/10.1109/ICSE-SEET.2017.20

[13] Pierre Flener. 2006. Realism in project-based software engineering courses: Rewards, risks, and recommendations. In *Proceedings of the International Symposium on Computer and Information Sciences*, Vol. 4263. Springer, Istanbul, 1031–1039. DOI: https://doi.org/10.1007/11902140_107

[14] S. A. A. d Freitas, W. C. M. P. Silva, and G. Marsicano. 2016. Using an active learning environment to increase students' engagement. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET'16).* 232–236. DOI: https://doi.org/10.1109/CSEET.2016.24

[15] Gerald C. Gannod, Janet E. Burge, and Michael T. Helmick. 2008. Using the inverted classroom to teach software engineering. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08).* ACM, New York, NY, 777–786. DOI: https://doi.org/10.1145/1368088.1368198

[16] K. Gary, T. Lindquist, S. Bansal, and A. Ghazarian. 2013. A project spine for software engineering curricular design. In *Proceedings of the 2013 26th International Conference on Software Engineering Education and Training (CSEET'13).* 299–303. DOI: https://doi.org/10.1109/CSEET.2013.6595265

[17] R. Green. 2008. Repetition and spacing effects. In *Learning and Memory: A Comprehensive Reference. Volume 2: Cognitive Psychology of Memory.* Elsevier, Oxford.

[18] E. Hill, P. M. Johnson, and D. Port. 2016. Is an athletic approach the future of software engineering education? *IEEE Softw.* 33, 1 (Jan. 2016), 97–100. DOI: https://doi.org/10.1109/MS.2016.15

[19] Google Inc. 2016. *Diversity Gaps in Computer Science: Exploring the Underrepresentation of Girls, Blacks and Hispanics*. Technical Report. Google, Inc. Retrieved fromhttps://services.google.com/fh/files/misc/diversity-gaps-in-computer-science-report.pdf.

[20] P. Johnson, D. Port, and E. Hill. 2016. An athletic approach to software engineering education. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET'16).* IEEE Computer Society, Dallas, Texas, 8–17. DOI: https://doi.org/10.1109/CSEET.2016.29

[21] B. R. von Konsky, M. Robey, and S. Nair. 2004. Integrating design formalisms in software engineering education. In *Proceedings of the 17th Conference on Software Engineering Education and Training 2004.* 78–83. DOI: https://doi.org/10.1109/CSEE.2004.1276514

[22] M. Kropp and A. Meier. 2014. New sustainable teaching approaches in software engineering education. In *Proceedings of the 2014 IEEE Global Engineering Education Conference (EDUCON'14).* 1019–1022. DOI: https://doi.org/10.1109/EDUCON.2014.6826229

[23] S. Kuhn. 1998. The software design studio: An exploration. *IEEE Softw.* 15, 2 (Mar. 1998), 65–71. DOI: https://doi.org/10.1109/52.663788

[24] A. J. Lattanze. 2016. Practice based studio. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET'16).* 1–7. DOI: https://doi.org/10.1109/CSEET.2016.40

[25] A. Meier, M. Kropp, and G. Perellano. 2016. Experience report of teaching agile collaboration and values: Agile software development in large student teams. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET'16).* 76–80. DOI: https://doi.org/10.1109/CSEET.2016.30

[26] M. Niazi. 2015. Teaching global software engineering: Experiences and lessons learned. *IET Softw.* 9, 4 (2015), 95–102. DOI: https://doi.org/10.1049/iet-sen.2014.0042

[27] Tom Nurkkala and Stefan Brandle. 2011. Software studio: Teaching professional software engineering. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11).* ACM, New York, NY, 153–158. DOI: https://doi.org/10.1145/1953163.1953209

[28] N. M. Paez. 2017. A flipped classroom experience teaching software engineering. In *Proceedings of the 2017 IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials (SECM'17).* IEEE, Los Alamitos, CA, 16–20. DOI: https://doi.org/10.1109/SECM.2017.6

[29] Pantelis M. Papadopoulos, Ioannis G. Stamelos, and Andreas Meiszner. 2013. Enhancing software engineering education through open source projects: Four years of students' perspectives. *Educ. Inf. Technol.* 18, 2 (Jun. 2013), 381–397. DOI: https://doi.org/10.1007/s10639-012-9239-3

[30] D. C. C. Peixoto, R. M. Possa, R. F. Resende, and C. I. P. S. Pádua. 2011. An overview of the main design characteristics of simulation games in software engineering education. In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET'11).* 101–110. DOI: https://doi.org/10.1109/CSEET.2011.5876076

[31] G. H. L. Pinto, F. F. Filho, I. Steinmacher, and M. A. Gerosa. 2017. Training software engineers using open-source software: The professors' perspective. In *Proceedings of the 2017 IEEE 30th Conference on Software Engineering Education and Training (CSEET'17).* 117–121. DOI: https://doi.org/10.1109/CSEET.2017.27

[32] Roger S. Pressman and Bruce Maxim. 2014. *Software Engineering: A Practitioner's Approach* (8 ed.). McGraw–Hill Education, New York, NY.

[33] R. Roshandel, J. Gilles, and R. LeBlanc. 2011. Using community-based projects in software engineering education. In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET'11)*. 472–476. DOI: https://doi.org/10.1109/CSEET.2011.5876127

[34] R. Simpson and T. Storer. 2017. Experimenting with realism in software engineering team projects: An experience report. In *Proceedings of the 2017 IEEE 30th Conference on Software Engineering Education and Training (CSEET'17)*. 87–96. DOI: https://doi.org/10.1109/CSEET.2017.23

[35] Evelyn Stiller and Cathie LeBlanc. 2002. Effective software engineering pedagogy. *J. Comput. Sci. Coll.* 17, 6 (May 2002), 124–134.

[36] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. 2008. Bloom's taxonomy for CS assessment. In *Proceedings of the 10th Conference on Australasian Computing Education (ACE'08)*. Australian Computer Society, Inc., Darlinghurst, Australia, 155–161.

[37] R. Thompson and W. Spencer. 1966. Habituation: A model phenomenon for the study of neuronal substrates of behavior. *Psychol. Rev.* 73, 1 (1966), 16–43.

[38] J. Vallino. 2013. What should students learn in their first (and often only) software engineering course? In *Proceedings of the 2013 26th International Conference on Software Engineering Education and Training (CSEET'13)*. 335–337. DOI: https://doi.org/10.1109/CSEET.2013.6595273

[39] J. Vanhanen, T. O. A. Lehtinen, and C. Lassenius. 2012. Teaching real-world software engineering through a capstone project course with industrial customers. In *Proceedings of the 2012 First International Workshop on Software Engineering Education Based on Real-World Experiences (EduRex'12)*. 29–32. DOI: https://doi.org/10.1109/EduRex.2012.6225702

[40] Gary N. Walker. 2004. Experimentation in the computer programming lab. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'04)*. ACM, New York, NY, 69–72. DOI: https://doi.org/10.1145/1044550.1041660 event-place: Leeds, United Kingdom.

Received November 2018; revised July 2019; accepted July 2019