2016 IEEE 29th International Conference on Software Engineering Education and Training

An athletic approach to software engineering education

Philip Johnson*, Dan Port†, Emily Hill‡,
*Department of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI USA
†Department of Information and Technology Management, University of Hawaii at Manoa, Honolulu, HI USA
‡Department of Mathematics and Computer Science, Drew University, Madison, NJ, USA
Emails: johnson@hawaii.edu, dport@hawaii.edu, emhill@drew.edu

***Abstract*—We present our findings after two years of experience involving three instructors using an "athletic" approach to software engineering education (AthSE). Co-author Johnson developed AthSE in 2013 to address issues he experienced teaching graduate and undergraduate software engineering. Co-authors Port and Hill subsequently adapted the original approach to their own software courses. AthSE is a pedagogy in which the course is organized into a series of skills to be mastered. For each skill, students are given practice "Workouts" along with videos showing the instructor performing the Workout both correctly and quickly. Unlike traditional homework assignments, students are advised to repeat the Workout not only until they can complete it correctly, but also as *quickly* as the instructor. In this experience report we investigate the following question: how can software engineering education be redesigned as an athletic endeavor, and will this provide more efficient and effective learning among students and more rapidly lead them to greater competency and confidence?**

***Keywords*-software engineering pedagogy, athletic training**

I. Introduction

Over many years of teaching software engineering, we have implemented innovations such as the flipped classroom, personal software process, simulated development environments (games), open source tools and projects, coding contests, and community projects. We hoped our efforts would move our students' experience from a dusty, dry subject to be endured into a course alive with possibilities. Based upon course evaluations, our prior approaches did appear to make our courses more interesting and enjoyable.

Yet there was something fundamentally dissatisfying when none of our students participated in a University of Hawaii Startup Weekend in 2013. This fast paced entrepreneurial workshop was in desperate need of technically competent developers. Why weren't our students interested? One reason stood out - they did not feel competent enough in their development skills to participate. How could this be? We just spent an entire semester on software development! There were other signs that something was amiss. Some of our students did poorly in subsequent courses that depended on their development skills. Many of our graduates decided not to pursue a career in software engineering. Certainly a career in software development is not for everyone, but we expect those students who have successfully complete our curricula to be at least interested in trying it out.

We now believe that our students' success depends greatly in their ability to develop competancy, and perhaps as importantly, a sense of confidence in their software development skills. Despite having implementing many of the advances and innovations suggested in the literature, we had not seen much improvement in these two key areas. Perhaps this is because our prior approaches did not incentivize students to internalize the mechanics of software development: skillful use of the languages, tools, and technologies so that their attention can focus on the actual experience of software engineering.

In this paper, we present our initial experiences confronting this educational challenge through an "athletic" approach to software engineering education (AthSE), which co-author Johnson developed in 2013, and which has been adapted by co-authors Port and Hill to their own software courses. AthSE is a pedagogy in which some or all of the course is organized into a series of skills to be acquired. For each skill, students are given "workouts" (practice problems) along with videos showing the instructor performing each workout correctly, quickly, with verbal explanations of their choice of tools and design techniques as they apply them. Students are advised to repeat the workout until they can complete it not just correctly, but in approximately the same time as the instructor.

For example, consider a workout in which the instructor creates a branch of a git repository, extends a feature, writes a unit test, commits the change, and merges the branch in 16 minutes. A student might give up on their first attempt after 40 minutes, then watch the instructor's video, then succeed on their second attempt in 30 minutes. On their third attempt, they succeed in less than 20 minutes, finally demonstrating the ability to solve the problem correctly and with approximately the same efficiency as the instructor.

The AthSE pedagogy was inspired by the observation that athletics, in contrast to traditional software engineering education, views time as a constrained resource, and well-trained athletes generally do not suffer from distraction or lack of confidence during training or competition. Even the "flow state"[1], revered among programmers as a rarely achieved kind of software development satori, is commonplace and unexceptional among athletes during competition. After two years of experience with three different instructors

2377-570X/16 $31.00 © 2016 IEEE
DOI 10.1109/CSEET.2016.29

using AthSE, we have observed a number of intriguing results, but also a few unexpected challenges:

- A very high percentage (often 100%) of the students in a class prefer "workouts" to the conventional classroom style, although they often get initially discouraged if they are not experiencing success.
- All three instructors who have used this pedagogy prefer it, although it requires significant work to design and implement workout videos demonstrating "good form".
- Successful students repeat at least some workout assignments multiple times. This contrasts with traditional homework assignments, which are virtually never repeated.
- Athletic software engineering seems to solve the "multitasking" or "divided attention" problem [2]. While working on class material, students do not check Facebook, email, text messaging, etc. They are in the "flow state", just as an athlete in a 100 yard dash is in the "flow state". The classroom becomes a very active, yet focused environment.
- Many students leave the class with higher competence and greater confidence in their development skills as compared to previously non-AthSE based classes. Fewer students fail or do poorly in the course.
- Many students become enthusiastic about the curricula, are willing to participate in extra curricula activities (e.g. Start Up Weekends), and generally appear to perform better in subsequent classes that depend on development skill. Students also appear to develop greater comradery.

While we have gathered a substantial amount of empirical data on results and have investigated theoretical underpinnings, the aim of this paper is to generate interest in experimenting with this approach to software engineering education. All materials and assessments discussed here are publicly available and the authors enthusiastically support use and adaptation of them by the software engineering education community.

II. Athletics as a Pedagogical Goal

We have a confession to make. For over 20 years, we've been teaching "cubicle" software engineering. This does not mean that we teach students to use punch cards, COBOL, and the waterfall lifecycle model. To the contrary, our curriculum appears quite modern: agile development processes, a "flipped" classroom [3], and modern tools and technologies including GitHub, Heroku, Bootstrap, and the like.

Cubicle software engineering refers to two pedagogical decisions we have not changed in 20 years:

1) A software development time frame measured in days, weeks, and months.
2) No explicit focus on the actual speed of coding.

Cubicle software engineering reflects what we believe is general industry practice. Consider a popular modern methodology, such as Scrum. Each day begins with a stand-up meeting, where developers report on the development tasks that they have, and haven't, accomplished. Based on this information, the team reassesses their priorities, and over time derives an estimate of their development "velocity". Developers are free to work as fast or slow as they want; management is prohibited from complaining about their "speed".

In the past few years, a different style of development has emerged within the context of hackathons and startup weekends. The relevant characteristic of these events is that they take place over 24 to 48 hours, and thus a "slow" developer is a significant liability to the team. In industry, where the development time-frame is measured in weeks or months, a "slow" developer could potentially compensate by working a few extra hours on nights or weekends to increase their "effective" speed. During a startup weekend, on the other hand, there are no "extra" hours, and slow developers are just plain slow.

Co-author Johnson looked at the projects created during a Honolulu Startup Weekend, and realized that the tools and technologies he was teaching provided an excellent basis for success. But what he wasn't teaching is how to use these tools to code *efficiently*: how to start with nothing and create a functional and interesting web application in a matter of hours. In fact, he taught the opposite, giving students significantly more time than required in order to remove time as a factor. For example, he might give students a week for a programming assignment that might take an instructor an afternoon to finish. As a result, Johnson found that his students were intimidated by Startup Weekend. The very idea of coding under time pressure fell completely outside their software development experience and training. So when he offered it as an alternative to a midterm exam, no students opted for it.

We now believe that in order for students to feel competent and confident enough to participate in a startup weekend or hackathon environment, they need to train for it. And this means not just learning useful languages, technologies, and design patterns, it also means learning to code efficiently. It means instead of thinking of development in terms of days and weeks, students must think in terms of hours and minutes. Instead of giving students 100x the time required, we should regularly put them into situations where they have just enough time to finish. In other words, students need to engage in software development as an athletic activity, not a cubicle activity.

Is athletic software engineering the right educational goal?: It is not unreasonable to question whether or not an athletic approach to software engineering, particularly the pursuit of speed in coding, is an appropriate pedagogical goal. As noted above, it is not part of mainstream software

development culture, and the most obvious measure of coding speed, LOC/hour, is a canonical example of measurement dysfunction[4]. So before detailing AthSE, here are three key educational challenges and objectives:

(a) **Speed does not mean sloppy, it means fluency.** There is a perception in software development that people who code fast are cutting corners and producing low quality work. Athletic endeavors are the opposite: the best athletes are fast precisely because they have the best technique and greatest efficiency of movement; in short, fast implies high quality.

Teaching students to code quickly in AthSE teaches them to be fluent with their tools and technologies. Because making errors slows development, they learn to avoid making errors, and to more efficiently find and fix the errors they make.

(b) **The flow state as normal state.** The programming culture reveres the "flow state" as a semi-mystical occurrence where deep, uninterrupted concentration makes time pass quickly, banishes all distractive thinking, and allows bursts of creativity. It is viewed as an elusive and transitory phenomena.

In AthSE, the "flow state is the normal state": give students a programming problem and very little time to solve it, and they will enter the flow state and stay there until they either finish the problem or run out of time.

(c) **Solving the multi-tasking problem.** A modern educational problem is multi-tasking: there is mounting research evidence that multi-tasking impairs learning[2].

Athletic software engineering education solves the multi-tasking problem by (i) creating a sense of urgency which (ii) creates the "flow state" (see above) which (iii) removes the desire to multi-task.

III. Athletic software engineering in a nutshell

To understand what makes our approach "athletic", it helps to first consider some common alternatives to software engineering education.

Lecture-based survey. One traditional educational model involves the use of one of the many high quality Introduction to Software Engineering textbooks, with lectures presenting material from various chapters, and tests that assess the ability of students to define or manipulate software engineering concepts such as the "Spiral Model", "White Box Testing", "Extreme Programming", etc. This approach has the benefit of being both comprehensive and consistent in the way students each semester encounter the material. However, it treats software engineering material at a conceptual level which does little to help build competence and confidence in applying these concepts.

Project-based practicum. This approach involves solicitation of "'real-world" application requirements from the surrounding community. Students form teams and attempt to build software to satisfy their customer needs. The project-based practicum tends to produce more engagement among many students, although the experiences encountered by each group varies a great deal based upon their community sponsor. In addition, the experience of a student within a single group can vary, and it is difficult to guarantee or assess that all students in the group are gaining the same software engineering experiences. The instructor must somehow find a balance between micro-managing the teams to ensure a successful outcome, or else let them learn their lessons the hard way.

Flipped classroom. A third approach is to "flip" or "invert" the classroom. In this case, lecture material is recorded and provided to students via YouTube, leaving class time for more active learning opportunities. For example, class time can be devoted to what was traditionally "homework", which is why this approach is known as "flipped". This approach requires students to focus on videos outside of class, which is highly susceptible to distraction and procrastination. There is little to motivate the student to actively watch the videos or even watch them at all.

Each of these approaches has their potential use cases, and aspects of each can be blended into a single course, but none attempts to *explicitly* address the problems of building competence and confidence in developing software.

Athletic software engineering education resolves this dichotomy by differentiating between the creative aspects and the mechanics of each software engineering skill to be taught. Almost by definition, it is impossible to define the "minimal" time for the creative part. However, as we have discovered, it is straightforward to specify a reasonable minimal time for the "mechanics", and that this creates an educationally interesting opening for application of athletic concepts.

Let's take a simple example: writing a unit test. In a lecture-based survey course, students might read a chapter about unit testing and learn how to compare and contrast it with other kinds of testing (integration testing, load testing, etc.). The instructor might require students to demonstrate the ability to express this conceptual knowledge on a written exam.

In a project-based practicum, students might be required to develop unit tests for their application. Different groups might develop their tests at different times and with different technologies.

In a flipped classroom, students might learn about unit testing through video lectures at home, then come into class and develop a few unit tests under the guidance of the instructor.

In athletic software engineering education, the writing of a unit test combines creative decisions (deciding what to test and why) and mechanics (the set of tasks to reify those decisions in high quality software). In the first author's software engineering classes, the mechanics involve testing a Java abstract data type using the JUnit library with code edited using the IntelliJ IDEA interactive development

environment. Adherence to coding standards is verified by Checkstyle, and the completed unit test is stored in a GitHub repository.

More specifically, mastering the mechanics of unit testing means the student can efficiently: sync their local repo with GitHub; create a local branch to hold their unit test development code; use IDEA shortcuts to create the Java class to hold the unit test and automatically import the appropriate JUnit library; apply refactoring if needed to extract a method for testing; use method completion to reduce the keystrokes required to create assertions; invoke the unit test within IDE; invoke Checkstyle to verify coding standards and fix any errors that occur; commit the finished code to the branch; and merge the branch into master.

As you can see, the mechanics involved with developing even a simple unit test are extensive, and involves an interplay between six languages, tools, and technologies (Java, JUnit, IntelliJ IDEA, git, GitHub, and Checkstyle). And, as we have discovered, students can be incapable of developing unit tests, or take an excessive time to do so, not because of the creative decisions, but simply because they do not have mastery of the mechanics and no amount of googling can rescue them.

The good news is that by integrating athletic concepts into the curriculum, students can not only gain mastery of these mechanics, they can gain this mastery in a way that also overcomes self-doubt in their ability to develop software. In a nutshell, athletic software engineering education involves the following:

(a) Structure the curriculum as a sequence of skills to be mastered, not as concepts to be memorized. While it is important, for example, for students to understand the conceptual difference between unit testing and load testing, athletic software engineering focuses on skills (i.e. mechanics) whose acquisition can be demonstrated via the solving of problems whose minimal time to solution is between 5 and 20 minutes. One of our courses teaches software engineering concepts through two tier web application development, with approximately a dozen "skills" each taking approximately a week to cover.

(b) Create a set of "training problems" for each skill, each accompanied by a video that demonstrates their solution in "optimal" time. Once a skill has been identified, the instructor provides background readings about the skill. More importantly, the instructor also provides sample problems whose resolution requires use of the skill, along with an online video (typically YouTube) that shows a timed, "reference solution" for the problem. The video solution time becomes the operational definition of "minimal" time (we call it "Rx" time) to solve that problem. For example, the Rx time for one of our unit testing sample problems is 15 minutes. In addition to Rx time, we also provide a "DNF" (Do Not Finish) time, which indicates the maximal amount of time to spend solving the problem before we recommend they simply start over. For the unit testing problems, DNF time was 20 minutes.

(c) Provide time to learn to solve the training problems in Rx time. Given the background readings and the sample problems, the students now must practice the mechanics until they can also solve the sample problems in close to Rx time. In a recent class, all but one of the students reported that they attempted the problems at least two if not three times in order to solve them in Rx time.

(d) Test mastery of the skill through an in-class, timed problem. To assess progress toward mastery, test students on a new problem requiring the skill that they have not seen before. Prior to class, the instructor must solve the problem to determine Rx time, and then adds 50-100% of that time to determine DNF time. For students to get credit for the skill, they must solve the problem both correctly and prior to the DNF time being reached. In the unit testing example, the in-class problem Rx time was 10 minutes, and the DNF time was set at 20 minutes. (About a quarter of the class DNF'd in a recent semester, either because they did not finish on time or did not produce a correct solution.)

(e) Move on to the next skill, typically based upon many of the same tools and technologies. Note that the unit testing skill was based upon six underlying technologies, so it's possible to leverage the learnings from one skill in surprising way. In this example course, the skill following unit testing was basic UI design, which did not use Java, JUnit, and Checkstyle, but did use IntelliJ, git, and GitHub.

The website for Advanced Software Engineering [5], held at the University of Hawaii at Manoa during Spring 2015, provides a complete example of athletic software engineering applied to a variety of skills.

In a nutshell, athletic software engineering education requires students to demonstrate mastery of the mechanics of various software engineering skillsets via timed assessments that they must complete both correctly and within a certain time limit. We have found that this reduces distraction, improves focus, and makes learning more efficient thereby building both competence and confidence in development skill.

IV. Implementing athletics in the classroom

Implementing an effective approach to athletic software engineering education in the classroom is difficult. Here are the key implementation concepts used by Johnson during three semesters of AthSE at both undergraduate and graduate levels and by Port in three undergraduate web application programming courses:

Workouts, not classes. Reframe expectations for classroom hours as "workouts", not "classes". Students do not come to class to passively listen to the instructor imparting information. Instead, they come to class prepared to engage in structured activities intended to assess and improve their ability to develop software quickly. This leverages development of

flipped classroom techniques for software engineering where acquisition of the material and "lectures" are designed to be seen online prior to class, not during it. Class time is focused on addressing gaps in understanding, applying lecture material, and mastery of skills.

WODs: the new normal. A standard component is the "Workout of the Day" or WOD (a term borrowed from CrossFit). This is a timed programming task performed by all members of the class with a simultaneous start time and a recorded finish time. Regular WODs are extremely important to developing speed, assessing competence, and building confidence. The presence of others simultaneously attempting to complete the same task in as short a time as possible creates motivation as well as a sense of camaraderie though shared trials and experience. Students are expected to "train" for WODs outside of class by repeating a practice WOD that addresses a similar software engineering skill. To determine the finish time for the practice WODs, and to provide a reference solution, screencasts are provided demonstrating the instructor solving the problem. The screencasts enable students to see a solution within a fixed amount of time and to learn how to efficiently solve a particular problem, often learning new skills along the way. DNF indicates a "point of no return". For homework assignments (practice WODs), reaching DNF indicates they should stop trying to solve the problem and watch the reference solution, then delete the work from their previous attempt, then try again. For in-class WODs it indicates that the student did not adequately train for that particular workout session.

In-class WODs are not graded according to the order in which students finish. Rather, 100% credit is awarded if the task is completed correctly within a fixed time limit (typically 10-30 minutes, depending upon the task). Students earn no credit if they do not complete the task correctly or fail to finish within the time limit. It's important to emphasize that WODs are not veiled exams or quizzes meant to torture them into learning the material. They are an assessment tool by which students can directly measure their progress and effectiveness in building their development skills. Repeatedly DNF'ing indicates that a student may need to review and adjust how they are training. For example, do they repeat a practice WOD until they are able to complete it within the expected time or do they not time themselves and assume that if they got it to work that they have mastered the task? WODs turn out to be interesting, fun, and motivating for students, and as a result, build their competence and confidence to the point where opportunities like Startup Weekend or Hackathons seem attractive to them as a place where they can show off their newfound skills.

Scoreboards for motivation, not humiliation. Publicizing a standard scoreboard, with names (or even pseudonyms) along with WOD performance creates a significant risk of humiliation for the slow students, who will be obvious from the public nature of the races themselves. On the other hand, providing feedback on how you are doing relative to others provides motivation to improve.

To minimize humiliation while providing motivation, we do not provide public data on individual performances, but rather the aggregate number of students who satisfied the WOD at various levels and some general discussion on the results without individual criticism.

Individual and group-based WODs. WODs can be designed for individuals or groups of students. Groups (generally pairs of students) are randomly assigned by the instructor; students cannot "cherry pick" partners in order to improve their performance.

Learning the skill is separate from performing the skill. It is not useful to give students a task for which they lack the required skill set and then time their completion of the task. For example, assume the task is to create a github repository, push a webapp's code into the repository, then set up continuous integration and deployment of that system using a cloud-based service. If the student has never done that task before, it could take many hours of research to figure out all of the steps and perform them successfully for the first time. On the other hand, this same sequence of steps is a canonical pattern that an experienced developer could accomplish in minutes given fluency with all of the associated technologies (i.e. in the case of one class, the technologies include Java, Git, GitHub, Play Framework, JUnit, Eclipse, Jenkins, MySQL, and CloudBees). There is easily a 100x difference in speed between novice and expert on this task.

To separate "learning" from "performing", the class separates "homework" from "workouts". Each workout (WOD) is prefaced by assigned homework and practice WODs. The students are provided reference material, lectures, and sample tasks they can use outside of class to learn how to perform the task and one or more practice WODs to "train" on this task. Then, the in-class WOD assesses their ability to complete the task in an efficient, rapid, and correct fashion. An example practice WOD is shown in 1. Note at the top of the practice WOD it refers to materials expected to have been learned prior to attempting the task.

Importance of skilled coaching and individualized feedback. AthSE demands significant skill and expertise on the part of the instructor. First, the instructor must be able to develop an appropriate sequence of homework assignments and in-class workouts (WODs and other exercises such as warm-ups for WODs). Second, the instructor must determine the Rx times. Third, the instructor should be able to monitor students and provide feedback on how they can improve their performance on WODs over time.

Startup weekend (or hackathon) or ambitious project as final exam. It would be ridiculous to give students a written final exam; that's not what they've been spending the semester training to accomplish. Instead, the course should include a startup weekend, hackathon or an ambitious, real-

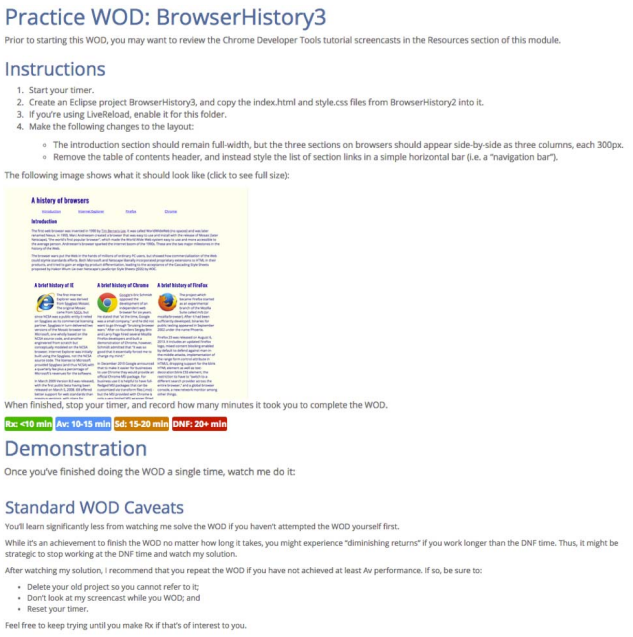

Figure 1. Example practice WOD. Rx means "expert time", Av means "average time", Sd means "standard time", and DNF means Does Not Finish

world final project under a tight time-frame (e.g. days not weeks) can be used. The idea is to provide students an opportunity to showcase their skills and leave with tangible evidence of their development competency. This not only builds their confidence for efficient programming in future classes, but leaves them with a development experience they can refer to in job interviews and scholarship applications.

Athletic software engineering education cannot be MOOCed or used in a large classroom setting. Massively open online courses are an important breakthrough in education, which can make certain types of learning available at scales previously unattainable. Athletic software engineering is probably not amenable to a MOOC environment or very large enrollment courses. Similar to athletic training, very few people have the internal motivation to "train" alone and in physical isolation from others. Development of athletic software development skills probably cannot happen without a group setting, excellent "coaching" by the instructor, and face-to-face interaction with coaches (instructor, TA, etc.) and other students.

In the next section, we present initial findings from our use of these techniques in a variety of settings along with student feedback.

V. Experiences Using ASE

The athletic approach described above has been used for over two years, in three different software development based courses, and with three different instructors. Here we summarize findings from implementing ASE in three software engineering courses by co-author Johnson, for two web-application development courses by co-author Port, and one elementary programming class by co-author Hill.

A. ASE in software engineering curriculum

Co-author Johnson has so far taught software engineering in an athletic style to two undergraduate software engineering classes and one graduate software engineering class. The following results are taken from these courses with a total of 46 students. To assess the approach, he required students to write technical essays on their progress through the course and administered a questionnaire near the end of the semester that obtained opinions from all students in both courses.

A brief summary of the significant findings include the following:

Students like the athletic approach. Out of 46 students surveyed, all but two (96%) prefer athletic software engineering to a more traditional course structure. One student commented, "I would choose to do WODs over the traditional approach because it helps you to become accustomed to working under pressure. I find myself learning more this way due to having to remember what I've done rather than searching up on how to do something and then forgetting soon after."

Students will redo training problems to gain skill mastery. While athletic software engineering makes it possible for students to repeat training problems if they do not achieve adequate performance, it does not mean students will do that in practice. Think back to your own scholastic endeavors: did you ever redo a home assignment from scratch just to see if you could finish it faster? One of fascinating findings is that the majority (72%) of students found it useful to repeat the training problems, and most repeated over half of the training problems at least once.

The athletic approach improves focus. Most students (82%) indicated that they believed that athletic software engineering helped improve their focus while learning the material. One student commented, "Like many students, when I do work at home I get distracted easily. This makes my time management skills very ineffective at times, and I would waste a lot of time [...] WODs definitely helped me to accomplish more in less time."

The athletic approach creates "comfort under pressure". Pressure is a part of a software developers life; starting at the interview which typically requires the applicant to solve a programming problem. Over 80% of the students felt that this approach helped them to feel comfortable with programming under pressure. One student commented: "I am indeed more confident in programming under pressure. I have learned to think not more quickly, but more calmly and collectively, as that is probably most important in completing a task faster."

More specifically, 80% of the students surveyed felt:

- WODs are preferable to traditional course structure.
- WODs help them to be more focused.
- An in-class, graded WOD is helpful.
- They became more comfortable with programming under pressure.
- They became more confident about their programming abilities.

In summary, initial results from student self-assessment of athletic software engineering as implemented by co-author Johnson indicates they prefer this style of education, they are motivated to practice skills repetitively to improve their efficiency, they believe the pedagogy improves focus, and they acquire a level of comfort with pressure during programming tasks.

B. AthSE in a business school curriculum

Co-author Port adapted the athletic software engineering approach to an introductory web applications programming course which serves as the entry course to the Management of Information Systems (MIS) major within the Shidler College of Business. While this course shares many similar objectives, goals, and challenges with typical software engineering courses, there are some important differences. For example MIS students are unlikely headed for a software engineering career and as such they will not need a high degree of technical depth in software development. Yet they likely will be involved in some way with the acquisition, development, maintenance or management of software systems and thus must be exposed to fundamental software engineering concepts. The majority of our MIS students will have little or no programming skills prior to this course, and they are unlikely to take another programming based course in the future. Nonetheless, subsequent core courses such as systems analysis and design will assume they have programming competency. The challenge of this course is to educate absolute novices to (1) acquire basic programming fluency, (2) acquire skills and strategies for becoming efficient in all phases of planning, designing, programming, documenting, and testing software applications, (3) understand why and where MIS people need the aforementioned. This must be accomplished all within a single semester.

Owing to these challenges, our interest in the athletic approach lies in rapidly building competence, and perhaps more importantly, having *confidence* in developing software. Lack of confidence has been a serious problem for our MIS students. It has led to a number of unpleasant consequences downstream such as poor performance MIS courses, a high incidence of cheating, and aversion to programming tasks and projects e.g. getting others to do the programming. This can lead some students to a lack of enthusiasm for MIS or dropping out of the major i.e. *"if this is what MIS is about I don't want to do it."*

Previously Port had implemented, with limited success, many of the modern approaches discussed in section III. His experience over the past three semesters with the athletic approach indicates that it is highly effective in rapidly building both competence and confidence in software development for extremely novice MIS students. Unexpectedly the athletic approach also appears to *generate enjoyment and enthusiasm* for building software after competence and confidence is achieved. In particular, it fosters determination in getting software to work and elation when it does, rather than fear and despondence when it does not.

Port applies athletic software engineering with WODs that focus on basic programming concepts (e.g. loops, arrays, etc.) rather than particular development technologies or software engineering concepts. There are 7-9 WODs used to solidify the application of a particular programming concept after it has been introduced and experienced from in-class lab exercises. WODs also do not persist for the entire class. They are primary used to rapidly build basic programming confidence. Practice WODs solutions are an excellent opportunity to introduce good programming practices, tools, and efficiency techniques in a natural hands-on way that students can immediately apply and appreciate e.g. using regular expressions to search and replace a pattern of variable names. WODs continue until the majority of the class are not DNFing. From the experience over two classes this occurs after 5-6 WODs. It's important to have a few WODs after this point to enable students to recognize and appreciate their improvement efforts to successfully complete a WOD. After basic programming concepts are covered, students switch to designing and implementing full applications of moderate complexity.

As prescribed by the athletic approach, throughout the class students are asked to write about their experiences and progress. This is essential in helping them identify challenges and recognize improvement. This is also a handy way to evaluate the effectiveness of the athletic approach. Their first essay summarizes their experience in attempting two HTML/CSS practice WODs. For each practice WOD they are asked describe how long it took to finish and what they learned (e.g. about HTML, CSS, Netbeans, LiveReload, or Chrome Developer Tools). For each WOD that they DNF'd, explain what happened and what was learned. If they did a WOD multiple times, report the times of all attempts and what was learned by doing them more than once. It is important that this essay is done prior to the first in-class WOD to set a baseline they can compare with later in their effectiveness of preparing for a successful WOD. After they experience two subsequent in-class WODs they are asked to write an essay discussing their experiences in performing WODs. What worked well, what they stumbled on, what was ineffective. They describe how practice WODs help to prepare them and what they could have done to be better prepared. After the in-class WODs end, students are asked

to evaluate themselves in regards to their programming skill, enthusiasm for programming, training using practice WODs, ability to complete labs, and what has helped or hindered their learning. These surveys, the course evaluations, and monitoring enrollments in subsequence MIS courses indicate:

- Students like the practice WODs and feel they learn a great deal by attempting them then watching a solution video.
- Some students do not like in-class WODs and are frustrated when they repeatedly DNF. They do not like the "speed-game" and highly constrained environment. However they eventually learn how to succeed and believe WODs are essential for building their programming competence. They do not believe WODs should be eliminated from the class or replaced with more traditional, outside of class assignments.
- Running WODs until students no longer DNF builds confidence and enthusiasm. After completion of the WODs, students felt ready to take on the challenge of building full applications with more complexity and less guidance.
- Compared to previous classes, students who experienced the athletic approach performed better and a higher percentage of students performed successfully in subsequent MIS courses that depend on development skill.
- Fewer students are dropping MIS as a major and there has been a notable increase in the number of entering the major.
- Students who were initially fearful of or dreaded programming had shifted their views ranging from neutral to enthusiastic. All students believe their development skills greatly improved, typically well beyond their expectations.
- The athletic approach can be discouraging to some students. They are disappointed or worried when they find their approach is not as effective as examples or other students.

C. AthSE in introductory programming

In 2015, co-author Hill deployed a modified athletic curriculum for introductory programming in CS 1 (python) and CS 2 (Java). The in-class, timed problem was assigned as homework if the students did not finish. However, to receive an A grade, the assignment must have been correctly completed during class. Grading preference was given for correctness over submitting on time. The classes were not completely flipped, but instead blended traditional lectures, code demos to problems such as previous WOD or practice WOD exercises, WODs, and in-class project work time. The CS 1 class focused on intraprocedural programming (ifs, loops, functions, etc.) within a single file, whereas the CS 2 course focused on object-oriented design and higher level design concepts in creating programs with multiple files (fields, methods, inheritance, abstract classes, interfaces, etc.).

Lecture format	CS 1	(%)	CS 2	(%)
traditional lecture	5	(28%)	3	(60%)
code demo	16	(89%)	4	(80%)
WOD	6	(33%)	0	(0%)
assignment work time	7	(39%)	5	(100%)
Total respondents	18		5	

Table I
PREFERRED CLASS FORMATS FOR CS 1 & 2 (STUDENTS COULD SELECT MULTIPLE LECTURE FORMAT TYPES).

1) Experience: Anecdotal feedback from students was positive for the practice WODs—many students said they liked learning from the videos and would sometimes request additional videos and practice WODs to help them learn difficult concepts. One student complained the videos were too long and meandering, since they walked through the thought process and steps taken by the instructor to solve the problem, rather than just jumping straight to the answer.

As an instructor, the videos provided additional content hours outside of class to further help struggling students. The downside is that creating the videos took additional prep time, but this time would be gained in future offerings of the course.

From a student learning perspective, the downside of the practice WODs is that students who ran out of time before the WOD would simply watch the video without trying it themselves first. They would be lulled into a false sense of confidence that would leave them ill-equipped to succeed on the WODs. Perhaps being more strict on enforcing the WOD time limits, or offering multiple WODs to demonstrate mastery of a skill would have prevented this.

In a current offering of the CS 1 course, we have simplified the WODs and scheduled them to occur during class *before* the practice videos are posted. Students still have a time limit on the practice WOD, but cannot see the video solution until after the in-class WOD. This forces students to do the practice WOD if they want to prepare for the in-class WOD, which is taken almost verbatim from the practice WOD. The practice WODs remain ungraded.

2) Student Feedback: Anonymous student survey feedback from the courses was mixed. In the CS 1 course, 18 of the 25 students registered responded to the survey, with two-thirds preferring the athletic approach over a more traditional style. In contrast, of the 20 students registered for the CS 2 course, only 5 students responded to the survey, and 80% preferred a more traditional approach. These students had taken a version of the CS 1 course the prior semester, with the same instructor, but with a more traditional homework and lab structure. Table I shows the type of class format students in each course preferred. Students tend to prefer

code demos, which is the category of the practice WODs, but they prefer them during class time. The CS 2 students who had a different course format with the same instructor in the prior semester preferred more traditional lectures and in-class work time, which were familiar activities to them from CS 1.

Students in both courses complained that the competitive nature of the WODs discouraged collaborative learning. A CS 1 student said:

> ... it created a hostile environment where people were afraid to admit that they didn't understand course material outside of class. Also, it made peers less likely to help each other or provide advice.

A CS 2 student felt:

> ... it wasn't realistic. In a true development team, there is more of a focus on collaboration and working together rather than a competitive focus.

Whereas other students found that the competitive nature spurred them to "do additional work using resources outside of the class."

When the students succeeded, it tended to create a significant confidence boost, although some students felt like success was only doing what they were "supposed to" and felt more defeated when they couldn't complete the WODs.

Students from both courses agreed that the athletic structure kept them focused, and really enjoyed the practice WODs:

> It was less stressful doing homework [practice WOD] because I knew that the homework was not graded. The homework was there solely to help me learn, and that absence of negative pressure allowed me to focus and concentrate more so than I usually do.

The observations from the introductory programming classes mirrored observations from the other experiences. Students focused on learning design and organization in a CS 2 class and an MIS course struggled with timed assignments, whereas students in a more traditional CS 1 or software engineering course with small, concrete assignments had a better experience.

VI. Discussion

An ongoing question for this approach to education is to better understand the kinds of material and situations in which AthSE is preferred over more traditional approaches. This is a difficult question to address: AthSE is a moving target that has evolved over its various applications as unexpected benefits or liabilities have been discovered. However we have amassed a great deal of anecdotal and empirical data such as WOD performance results, student self-reported assessments, course evaluations and class surveys. These help us better understand and define what we are trying to achieve.

Do students "get better" over time at software development as a result of this kind of pedagogy? Some insight into this may be gained by analyzing WOD results. But gaining empirical insight from WOD results into whether the students are improving in performance over time is challenging due to the following confounding variables:

- The WODs vary in difficulty, so an increase or decrease in WOD time does not necessarily indicate an increase or decrease in performance capability; it is more likely due to the difficulty of the task.
- The instructors assignment of Rx, Sd, and Av times is arbitrary. It is quite possible that increases in (for example) the number of Rx performances over time is not due to actual performance improvement, but merely due to assigning an easier threshold for Rx as the course goes on. Nonetheless, consider Figure 2 which shows the percent of students who DNF'ed on the chronologically ordered individual WODs.

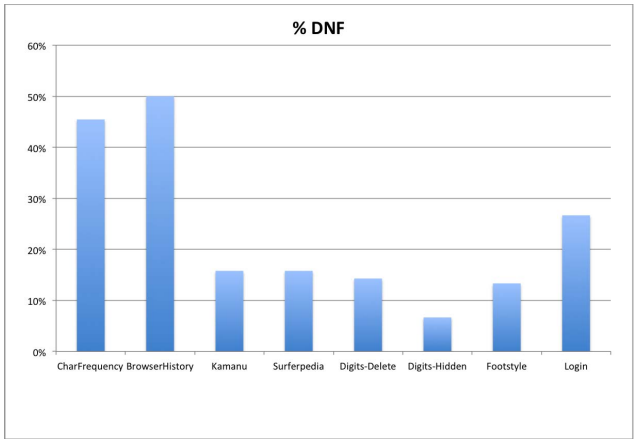

Figure 2. WOD DNF % over time ICS 314 Fall 2013

Clearly, the percentage DNF shows a precipitous decline after the first two WODs. We believe that the decline in DNFs are a result of: (a) students learning how to use the homework, including repeating practice WODs, to prepare for the WOD; (b) students becoming accustomed to "programming under pressure", and not having it impede their ability to accomplish the task at hand; and (c) several of the poorer performing students dropping the class over the course of the first six WODs. (Although note also that one of the highest performing students also dropped the class, so attrition did not occur from the bottom only.)

We believe that this decline in DNF cannot be attributed to the WODs becoming easier. The Digits-Delete WOD is substantially more complex than the CharFrequency WOD, even though the Rx times are the same.

One unexpected result we have observed from implementing an athletic approach is stronger student course evaluations. Students tend to feel the course provides high value and there is great appreciation for the instructor. See [6] for examples. Perhaps this is due to tangible nature of seeing WOD performance improve and the instructor as "coach" rather than adversary or task master.

AthSE appears to have a substantial impact on learning outcomes. Much of this impact appears to be quite positive, however there are some challenges. For example, fear of failure and WOD performance anxiety at the beginning of the course. However, this appears to abate as students become more comfortable with the approach.

One thing we can say almost with certainty about the future of software engineering education is that our students' success in their subsequent courses and in pursuing a software engineering career increasingly demands they are able to quickly learn apply the mechanics of software engineering competently and confidently.

Based upon our initial experiences, we believe an athletic pedagogy will find its place in the future of software engineering education as a way to help students efficiently acquire mastery of the mechanics of software engineering, and thus create additional temporal and mental space for the creative problem solving required in our discipline. As seen by the diversity of student responses to different adaptations, the approach is still in its infancy and will continue to be refined and improved with additional experience. We invite software engineering educators who find this approach of interest to join with us to move it forward.

REFERENCES

[1] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*, ser. Perennial Modern Classics. Harper & Row, 1990. [Online]. Available: https://books.google.com/books?id=V9KrQgAACAAJ

[2] A. M. Paul, "How does multi-tasking change the way kids learn?" Mind/Shift (online), http://ww2.kqed.org/mindshift/2013/05/03/how-does-multitasking-change-the-way-kids-learn/, May 2013.

[3] J. L. Bishop and M. A. Verleger, "The flipped classroom: A survey of the research," in *Proceedings of the ASEE National Conference*, Atlanta, Georgia, 2013.

[4] C. Kaner, "A short course in metrics and measurement dysfunction," http://www.kaner.com/pdfs/metrics_measurement_dysfunction.pdf, 2002.

[5] P. Johnson, "Advanced Software Engineering," http://philipmjohnson.github.io/ics613s15/, May 2015.

[6] ——, "Ics 314 fall 2013 course evaluations," http://www.hawaii.edu/ecafe/published-results.html?id=1912#191753.