Athletic Software Engineering Education: Preliminary Evidence

Emily Hill[1], Philip M. Johnson[2], and Daniel Port[3]

[1]Department of Mathematics and Computer Science, Drew University, Madison, NJ.
[2]Department of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI.
[3]Department of Information and Technology Management, University of Hawaii at Manoa, Honolulu, HI.

October 22, 2015

In the past 10 years, much evidence has accumulated regarding the negative effects of multi-tasking and continous partial attention on learning. One study found that students who multi-tasked spent only 65% of their time actively learning, that it took them longer to complete assignments, that they made more mistakes, decreased their ability to remember material later, and that they showed less ability to generalize the information they learned to new contexts [2].

Traditional educational approaches to software engineering education consisting of in-class lectures, unsupervised homework assignments, and occasional project deadlines are fertile ground for the distraction economy. What can we do to win back student attention and help them learn the skills they need for modern software engineering?

We present our initial experiences with an "athletic" approach to software engineering education, which co-author Johnson developed in 2013 as an experiment in software engineering education, and which has been adapted by co-authors Port and Hill in their own courses. The athletic software engineering pedagogy is inspired by athletic training, where time is viewed as a constrained resource that forces athletes to avoid distractions. Can software engineering education be redesigned as an athletic endeavor, and will this lead to more efficient and effective learning among students?

1 Athletic software engineering

Traditional approaches to software engineering education include lectures, projects, and flipped instruction modes. 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 distraction in modern educational environments. Put another way, how can we design the educational process to incentivize students to focus single-mindedly on the learning of complex, multi-step tasks without multi-tasking?

Athletic software engineering education attempts to answer these questions by adopting two simple features of conventional athletic training. First, the primary goal is to minimize elapsed

time. Many sports (such as running, cycling, etc.) are based upon completing a task in a minimal amount of time. Second, minimum elapsed time results from high quality effort. To go fast, you must have good form and technique.

Neither of these features are typically present in the software engineering classroom. First, assignments are generally structured to *eliminate* time as a constraint: for example, if the instructor believes a problem could be completed in a day, they might provide a week. This prevents students from ever claiming that they "didn't have enough time to finish". Second, in software engineering, working "fast" is generally viewed as working "sloppy". This is in direct contrast to athletic endeavors, where sloppiness generally produces slowness.

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, writing 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). 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.

The mechanics involved with developing even a simple unit test 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 the lure of technological distraction. In a nutshell, athletic software engineering education involves the following:

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

2. Create a set of "training problems" for each skill, each accompanied by a video that demonstrates their solution in "optimal" time.

3. Provide time to learn to solve the training problems in Rx, or "minimal" prescribed, time.

4. Test mastery of the skill through an in-class, timed problem, similar to a workout of the day (WOD).

5. Move on to the next skill, typically based upon many of the same tools and technologies.

The website for Advanced Software Engineering [1], 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.

To summarize, athletic software engineering education requires students to demonstrate mastery of the mechanics of various software engineering skill sets via timed assessments that they must complete both correctly and within a certain time limit. We claim that this reduces distraction, improves focus, and makes learning more efficient.

2 Evidence

The athletic approach has been evaluated in two software engineering courses by co-author Johnson, adapted to a business school curriculum by co-author Port, and adapted to an elementary programming class by co-author Hill.

2.1 Athletic education in software engineering

Co-author Johnson taught software engineering in an athletic style to both an undergraduate software engineering class in 2014 and a graduate software engineering class in 2015, with a total of 29 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 opions from all students in both courses. A brief summary of the significant findings include the following:

Students like the athletic approach. Out of 29 students surveyed, all but one (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. Surprisingly, 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. [...] WODs definitely helped me to accomplish more in less time."

The athletic approach creates "comfort under pressure". Pressure is a part of a software developer's life. Over 80% of the students felt that this approach helped them to feel comfortable with programming under pressure.

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.

2.2 Athletic education in a business school curriculum

Co-author Port adapted the athletic software engineering approach to an introductory web applications programming course to a Management of Information Systems (MIS) major. The challenge of this single semester course is to educate absolute novices to (1) acquire basic programming fluency, (2) acquire skills and strategies for becoming efficient in all phases of software development, and (3) understand why and where MIS people apply software development skills. Owing to these challenges, our interest in the athletic approach lies in rapidly building competence and confidence in developing software to increase students' future performance in MIS courses.

Our experiences over the past year 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. Curiously the athletic approach also fosters greater collaboration and students do not feel in competition with each other. To the contrary, they feel the drive to excel and help each other understand the material and master the assignments.

Through student comments and survey data, co-author Port summarizes student reaction to his adaptation of athletic software engineering as follows:

- Students like the practice WODs and feel they learn a great deal by attempting them and then watching a solution video.
- Students do not like in-class WODs and are frustrated when they repeatedly DNF. However they eventually learn how to succeed and believe WODs are essential for building their programming competence.
- 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 skills.
- 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.

2.3 Athletic education in introductory programming

Co-author Hill deployed an adaptation of athletic software engineering for introductory programming in CS 1 (python) and CS 2 (Java). The in-class, timed problems were assigned as homework if the students did not finish. However, to receive an A grade, the assignment must have been correctly completed during class.

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.

In a subsequent offering of the CS 1 course, we 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.

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. Unfortunately, in the CS 2 course, only 5 students responded to the survey, rendering the results uninterpretable. Unlike Port's students, students in CS 1 and CS 2 complained that the competitive nature of the WODs discouraged collaborative learning. For example, one 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." On the other hand, another students noted that the competitive nature spurred them to "do additional work using resources outside of the class."

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."

3 Conclusion

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] Philip Johnson. Advanced Software Engineering. `http://philipmjohnson.github.io/ics613s15/`, May 2015.

[2] Annie Murphy 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.