Athletic Software Engineering Education

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

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

May 26, 2015

1 Introduction

Startups in Silicon Valley and elsewhere have recently focused on *disrupting* education with online platforms such as Coursera, EdX, and CodeAcademy. What is less obvious is a parallel, implicit, and much better funded focus on *distracting* education through the rise of mobile applications intended to encourage multi-tasking and continuous partial attention. For example, one designer cites "design for distraction" as his Number 1 Principle [6], and others write of the rise of the *attention economy*, a "system that revolves primarily around paying, receiving and seeking what is most intrinsically limited and not replaceable by anything else, namely the attention of other human beings" [1]. This focus on distraction and attention has reaped dubious benefits: a 2013 Internet Trends report found that people check their phones an average of 150 times per day [4].

Why does this matter to the future of software engineering? It matters because our future software engineers are typically enthusiastic consumers of modern, mobile technology, and because this same technology is actively degrading their ability to do the focused, in-depth learning required to acquire advanced software engineering skills.

In the past 10 years, much evidence has accumulated regarding the negative effects of multitasking 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 [5].

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. Social applications such as Facebook and Twitter are designed to encourage continuous partial attention, but the impact of this distraction on deep learning is significant. What can we do to win back student attention and help them learn the skills they need for modern software engineering?

In this article, we present our initial experiences with an "athletic" approach to software engineering education, which the first author developed in 2013 as an experiment in software engineering education, and which has been adapted by the second and third authors to their own software courses. The athletic software engineering pedagogy was inspired by the observation that athletic training and subsequent competitions, in contrast to traditional software engineering education, view time as a constrained resource, and athletes generally do not suffer from distraction or continuous partial attention during training or competition. Even the “flow state”, revered among programmers as a rarely achieved kind of software development satori, is commonplace and unexceptional among athletes during competition. Can software engineering education be redesigned as an athletic endeavor, and will this lead to more efficient and effective learning among students? In this paper, we will present our findings and what they might imply for the future of software engineering education.

2 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 and the learning process is quite susceptible to modern technological distraction.

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 micromanaging the teams to ensure a successful outcome, or else let them learn their lessons the hard way (“This is what happens when you wait until the last minute!”)

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.

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? And if we can succeed, does that lead to more efficient and effective learning, which means that students will acquire more information in a given semester and retain it longer?

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, 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 recent software engineering class, the mechanics involved 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 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. While it is important, for example, for students to understand the conceptual difference between unit testing and load testing, athletic software engineeering 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.

2. **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.

3. **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.

4. **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.)

5. **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 [2], 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 claim that this reduces distraction, improves focus, and makes learning more efficient.

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

3 Findings

The athletic approach described above has been evaluated in two software engineering courses by the first author, adapted to a business school curriculum by the second author, and adapted to an elementary programming class by the third author. This section discusses some initial findings from all three settings.

3.1 Athletic education in software engineering

The first author 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 to obtain student opinions. 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 Ive 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."

In summary, initial results from student self-assessment of athletic software engineering 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.

Let's now take a look at results from some adaptations of this approach.

3.2 Athletic education in a business school curriculum

Dan writes about his experiences here.

3.3 Athletic education in introductory programming

Emily writes about her experiences here.

4 The future of software engineering education

One thing we can say almost with certainty about the future: technology will be more distracting than ever. If students are currently distracted by Facebook, Twitter, and YikYak, imagine the allure of 3D virtual reality environments such as the Oculus Rift. One thing we can also say: it is unlikely that the future will bring some educational silver bullet that makes learning software engineering quick and easy.

Based on our prior experience with serious game design [3], we do not believe that "gamification" holds a major role in the future of software engineering education. Providing leaderboards, awarding badges, and so forth can increase engagement to some degree, but we do not believe it is capable by itself of incentivizing the sustained, focus attention required to assimilate complex software development skillsets. We believe gamification works best for learning relatively simple, conceptual knowledge.

Another possibility for the future is that expansion of programming competitions such as TopCoder into the realm of software engineering. Such a development would be quite interesting as a way to provide visibility tohighly skilled software engineers, but such competitions are meant as an assessment vehicle, not for learning.

"Bootcamps" are a recent innovation for teaching coding skills; these courses require full-time residency and total commitment for six weeks to six months, and in many cases take complete novices and help them to become employable as web developers. It seems quite plausible that this model could be extended to software engineering with positive results: after all, complete immersion in any activity for six weeks to six months should yield significant progress. However, since most people cannot check out of their lives (and livelihood) for such time periods, software engineering bootcamps, if they arise, will be a niche approach applicable to only a few people.

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. This approach is still in its infancy and will certainly 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] Tomas Chamorro-Premuzic. The distraction economy: how technology downgraded attention. TheGuardian (online), http://www.theguardian.com/media-network/media-

network-blog/2014/dec/15/distraction-economy-technology-downgraded-attention-facebook-tinder, December 2014.

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

[3] Philip M. Johnson, Yongwen Xu, Robert S. Brewer, Carleton A. Moore, George E. Lee, and Andrea Connell. Makahiki+WattDepot: An open source software stack for next generation energy research and education. In *Proceedings of the 2013 Conference on Information and Communication Technologies for Sustainability (ICT4S)*, February 2013.

[4] Mary Meeker and Liang Wu. Internet Trends D11 Conference. KPCB (online), http://www.kpcb.com/blog/2013-internet-trends, May 2013.

[5] 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.

[6] Luke Wroblewski. 4 surprising app-design principles, from the instagram of quick quizzes. Fast Company (online), http://www.fastcodesign.com/1672257/4-surprising-app-design-principles-from-the-instagram-of-quick-quizzes, December 2013.