Editor: **Rafael Prikladnicki**
Pontifica Universidade Catolica do Rio Grande do Sul
rafael.prikladnicki@pucrs.br

Is an Athletic Approach the Future of Software Engineering Education?

Emily Hill, Philip M. Johnson, and Daniel Port

IN THE PAST 10 YEARS, there has been considerable evidence of the harmful effects of multitasking and other distractions on learning. One study found that multitasking students spend only 65 percent of their time actively learning, take longer to complete assignments, make more mistakes, are less able to remember material later, and show less ability to generalize the information they learned for use in other contexts.[1]

Traditional software engineering education approaches—in-class lectures, unsupervised homework assignments, and occasional projects—create many opportunities for distraction.

To address this problem, coauthor Philip M. Johnson developed an "athletic" software engineering education approach, which coauthors Emily Hill and Daniel Port adapted for use in their courses. We wanted to determine if software engineering education could be redesigned to be like an athletic endeavor and whether this would improve learning.

Athletic Software Engineering

We wanted to design the educational process to incentivize students to avoid multitasking and focus on learning complex, multistep tasks.

Athletic software engineering education adopts simple features of conventional athletic training. The primary goal is to minimize the time students need to accomplish a task. Many sports, such as running and cycling, are based on completing a task in a minimal amount of time. Another goal is to encourage a high-quality effort, which leads to better results.

Generally, neither feature is found in the software engineering classroom. Assignments usually eliminate time constraints. For example, if instructors believe a problem could be completed in a day, they might provide a week, thereby preventing students from claiming that they didn't have enough time to finish. Also, in software engineering, working quickly is typically viewed as working sloppily. This contrasts with athletic endeavors, in which sloppiness often produces slowness.

Athletic software engineering education resolves this dichotomy by differentiating between the creative aspects—for which minimum times can't be defined—and the mechanics—for which they can—of each skill to be taught.

Let's use writing a unit test as a simple example. In a lecture-based survey course, students might read a chapter about unit testing and learn how to compare and contrast it with integration testing, load testing, and other kinds of testing. The instructor might require students to express this conceptual knowledge via a written exam. In a project-based practicum, students might have to develop unit tests for an application. Different groups might develop their tests at different times and with different technologies. In a flipped classroom, students might learn about unit testing at home via videos and develop unit tests in class under the instructor's guidance.

In the athletic approach, unit-test writing combines creative decisions (deciding what to test and why) and mechanics (performing the tasks necessary to yield high-quality software).

The mechanics of developing even a simple unit test involve multiple languages, tools, and technologies. Students can be incapable of developing unit tests or take a lot of time to do so not because of their creative decisions but because they haven't mastered the mechanics. The good news is that by integrating athletic concepts into the curriculum, students can master these mechanics without experiencing distractions.

In a nutshell, athletic software engineering education involves

- structuring the curriculum as a sequence of skills to master, not concepts to memorize;
- creating a set of training problems for each skill, accompanied by a video demonstrating how to solve them in a minimal amount of time;
- providing the opportunity to learn to solve the problems in the prescribed amount of time;
- testing mastery of a skill through an in-class, timed problem, similar to physical training's workout of the day (WOD); and
- acquiring the next skill, typically by employing many of the tools and technologies previously learned.

Students agreed that the athletic software engineering education approach kept them focused.

The website for Johnson's Spring 2015 advanced software engineering class at the University of Hawaii at Manoa (http://philipmjohnson:github:io/ics613s15) provides a complete example of applying athletic software engineering to a variety of skills.

This approach requires students to demonstrate mastery of various software engineering skill sets' mechanics via assessments that they must complete correctly within a time limit. This reduces distraction, improves focus, and makes learning more efficient.

Evidence

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

Athletic Education in Software Engineering

Johnson used an athletic style to teach software engineering to an undergraduate software engineering class in 2014 and a graduate software engineering class in 2015. The two had a total of 29 students. To assess the approach, he required students to write technical essays on their progress and administered a questionnaire near the semester's end that obtained their opinions.

Of the students surveyed, all but one (97 percent) preferred the athletic course structure to the traditional one. A participant commented,

> *I would choose to do [academic] 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 for how to do something and then forgetting soon after.*

Athletic software engineering lets students repeat training problems if they don't achieve adequate performance. In our study, 72 percent of them found it useful to repeat the problems, and most repeated more than half of the problems at least once.

Of responding students, 82 percent said athletic software engineering improved their focus while they learned the material. One commented,

> *Like many students, when I do work at home, I get distracted easily. … WODs definitely helped me to accomplish more in less time.*

Pressure is a part of a software developer's life. More than 80 percent of the students said the athletic approach helped them feel comfortable programming under pressure.

Athletic Education in Business School

Port adapted the athletic approach to an introductory Web-application-programming course for management of information systems (MIS) majors. The challenge was to give novices basic programming fluency, skills and strategies for becoming efficient in all software development phases, and an understanding of why and where MIS workers use these abilities. We wanted to use the athletic approach to rapidly build competence and confidence in developing software to improve students' future performance in MIS courses.

Our experience over the past year indicates the athletic approach was highly effective in achieving these goals. Unexpectedly, it also generated enjoyment and enthusiasm for building software once the students achieved competence and confidence. In addition, it fostered both the determination to make software work and elation when it did, rather than fear and sadness when it didn't. Port's students said that the athletic approach promoted greater collaboration and that they didn't feel competition but instead wanted to help one another understand the material and master the assignments.

Students liked the practice WODs and learned a great deal by trying them and then watching a video of the solution. However, they didn't like in-class WODs and were frustrated when they repeatedly didn't finish them. Nevertheless, they eventually succeeded and decided that WODs were essential for building programming competence. Running WODs until students could finish them built confidence and enthusiasm. Upon completion, students felt ready to take on the challenge of building full applications with more complexity and less guidance.

Students who experienced the athletic approach did better than those whose classes took a more traditional approach, and a higher percentage performed successfully in subsequent MIS courses that depended on development skills. However, the athletic approach discouraged some students who didn't do as well as they expected or who weren't as successful as other students.

Athletic Education in Introductory Programming

Hill adapted the athletic approach for introductory programming classes in Python and Java. She assigned the in-class, timed problems as homework if the students didn't finish. However, to receive an A on an assignment, they had to correctly complete it in class.

Students said they liked working on the practice WODs and learning from the videos, and sometimes requested more of each to help learn difficult concepts.

Anonymous student survey feedback was mixed. In the Python course, 18 of the 25 students responded, with two-thirds preferring the athletic approach over a more traditional style. Unfortunately, in the Java course, only five of 24 students responded, rendering the results insignificant. Unlike Port's students, those in Hill's Python and Java classes complained that the WODs' competitive nature discouraged collaborative learning. For example, one said,

> *[I]t 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 student noted that the competition spurred them to "do additional work using resources outside of the class."

Both courses' students agreed that the athletic structure kept them focused and that they really liked the practice WODs. Said one,

> *It was less stressful doing [practice WODs] 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 than I usually do.*

Traditional software engineering education approaches create many opportunities for distraction.

Based upon our initial experiences, we believe an athletic pedagogy will find its place as a way to help students efficiently master software engineering's mechanics and better enable them to handle the creative problem solving that our discipline requires. As the diverse student responses to different adaptations showed, the approach is still in its infancy. We will continue to refine and improve it with additional experience and invite software engineering educators who find this approach of interest to join us.

Reference

1. A. Murphy Paul, "How Does Multitasking Change the Way Kids Learn?" *MindShift*, 3 May 2013; http://ww2.kqed.org/mindshift/2013/05/03/how-does-multitasking-change-the-way-kids-learn.

EMILY HILL is an assistant professor of computer science in Drew University's Department of Mathematics and Computer Science. Contact her at emhill@drew.edu.

PHILIP M. JOHNSON is a professor in and the associate chair of the University of Hawaii at Manoa's Department of Information and Computer Sciences. Contact him at johnson@hawaii.edu.

DANIEL PORT is an associate professor in the University of Hawaii at Manoa's Information Technology Management Department. Contact him at dport@hawaii.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

Agile Alliance®

PRESENTS

AATC

#AATC2016

Agile Alliance
Technical
Conference
April 7 - 9
2016
Raleigh, NC USA

Register Now
www.AgileAlliance.org

Featured Speakers

Martin Fowler
Woody Zuill
Neil Johnson
Maaret Pyhajarvi
Kate Falanga
Amital Schlair
Llewellyn Falco
Alex Schladebeck
Kent Graziano
Pat Maddox
James Shore
Matthew Dodkins
Fred George
Justin Searls
Melissa Perri
Jeff Morgan
Ardita Karaj
Cassie Shum
Sandi Metz
James Lewis
Michael Nygard
Seb Rose
Julie Lerman
Rebecca Wirfs-Brock
Bryan Beecham
James Grenning
Rachel Davies
Chris Kozak

Editor: **Philippe Kruchten**
University of British Columbia
pbk@ece.ubc.ca

Software Is Driving Software Engineering?

George Hurlburt and Jeffrey Voas

SOFTWARE ENGINEERING is quite well defined. In 2014, the IEEE Computer Society released the third edition of its comprehensive *Guide to the Software Engineering Body of Knowledge* (*SWEBOK Guide*).[1] Figure 1 shows part of the SWEBOK Guide's conceptual layout. The boxes show major topics, with subtopics listed in the descending structures. Each subtopic is further broken down and supported by even deeper levels, leading to the textual treatment of everything.

Despite the *SWEBOK Guide*'s thoroughness and apparent currency, it faces one fundamental challenge. Software continues to morph and expand in influence with increasing rapidity.

Why must the *SWEBOK Guide* face continual change? It turns out we're living in a physical world that's moving at the speed of software. This means that software's trajectory will drive software engineering, not vice versa.

A Brief History of Software

Consider the early software achievements, in which linear mathematics reigned supreme. Linear ballistic trajectory calculation was considered a triumphant achievement in the late '40s. The US space program brought ever more dynamic mathematical navigation problems to the forefront, literally taking us to the moon. Relational database management systems began to overcome expensive storage constraints and brought transaction processing to businesses, thus fueling functional programming by the '60s. PC-compatible operating systems brought computational power to individuals, incidentally vastly expanding the pool of potential programmers in the '80s.

Lately, the Internet has pioneered the notion of a global network in which everything can be connected. Now, the Internet of Anything is rapidly extending this notion well beyond human networks.[2] The huge mobile-device market is further reinforcing and hastening this network phenomenon. By 1989, the mobile phone packed more computational power than an Apollo mission computer.

Software has evolved similarly. Once considered a tool for rapidly and efficiently solving complicated mathematical problems, software has become a logical means to relate diverse ideas across vast networks. In so doing, software has migrated from mathematically precise expression to an environment in which meaning and data provenance often matter. It now supports expression of human abstractions understandable only in increasingly fuzzy functional contexts. Software can no longer be decoupled from the processes or functions it supports. As programming languages, such as Haskell, become more abstract, the question of precise meaning becomes increasingly urgent. Ontology is already becoming a prerequisite to disambiguation of semantic variation in which the relationships between plentiful software nodes are overwhelmingly many-to-many.[3]

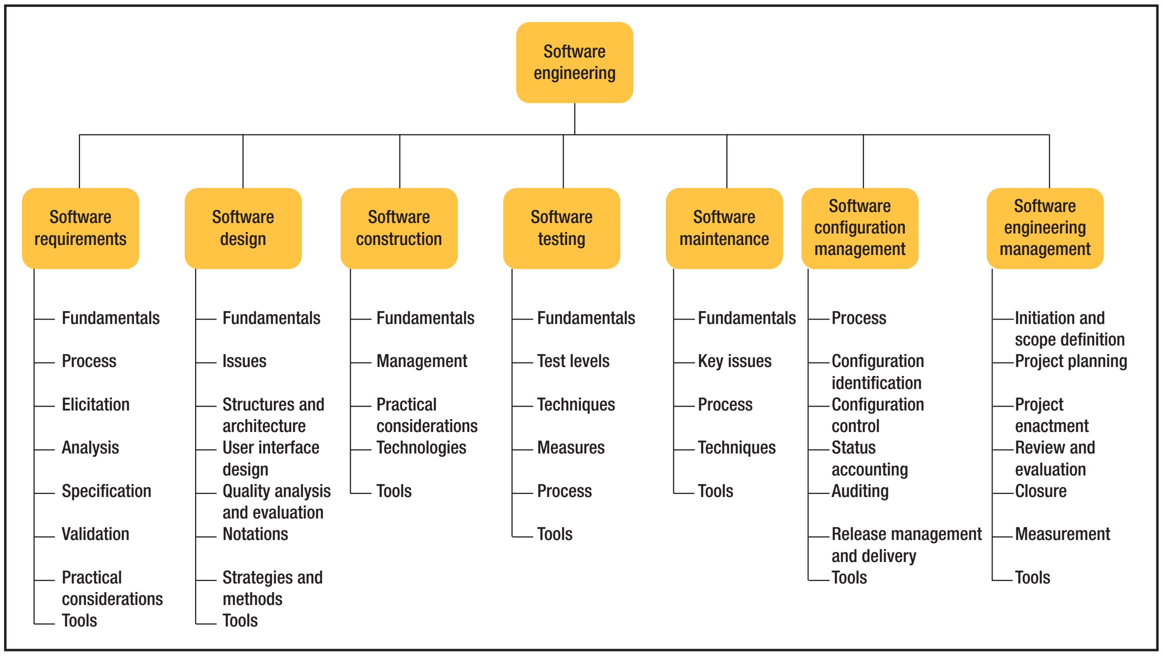

FIGURE 1. Part of the conceptual layout of the *Guide to the Software Engineering Body of Knowledge* (*SWEBOK Guide*). The boxes show major topics, with subtopics listed in the descending structures.

Software has transited from standalone programs performing singular functions to deeply embedded control mechanisms in vast system-of-systems environments. The epitome of such an environment is the smart grid, in which many key variables, typically outside the system, are in constant flux—sometimes somewhat rhythmic and sometimes totally asymmetric.

Because software routines are deeply embedded in systems, they too become highly interdependent. Today, any software interaction suggests that there are multiple paths, all influenced by sensor input from the external environment, to achieve a given end. Consider an autonomous automobile informed largely by a constantly learning Bayesian network. The optimal solution for a given subsystem at one microsecond might differ significantly from subsequent solutions in succeeding microseconds.

So, cause-and-effect relationships relate to paths through multiple software modules as influenced by sensor input and feedback, not by any single program's direct, discernable action. This argues against strict determinism, refutes reductionism as a valid software-testing approach, and drives any solution to nonlinear proportions. Indeed, software has moved standalone routines to adapt along with complex systems; in so doing, these routines have become complex adaptive entities in their own right. Embedded software's nonlinearity further refutes the notion that we can engineer software, much less test it, in any classically linear fashion.

As systems of systems become further embedded in networks of networks, the potential for self-organizing behavior increases substantially. Consider a network of autonomous vehicles on grid-enabled highways. The realm of nonlinear decision points and potential paths will grow to mammoth proportions as the Internet of Anything advances.

What Motivates Software Engineers?

For future software to be managed effectively, it would appear that dynamic software interdependence reigns supreme. But does this mesh with the nature of software developers, who live in the moment?

Monetary Gain

Some people would assert that money motivates. Software engineers are generally well compensated. According to the US Bureau