Using the Internet of Things to Teach Good Software Engineering Practice to High School Students

Christine Julien
The Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX USA
E-mail: c.julien@utexas.edu

Abstract

This paper describes a course to introduce high school students to software engineering in practice using the Internet Of Things (IoT). IoT devices allow students to get quick, visible results without watering down technical aspects of programming and networking. The course has three broad goals: (1) to make software engineering fun and applicable, with the aim of recruiting traditionally underrepresented groups into computing; (2) to make young students begin to approach problems with a design mindset; and (3) to show students that computer science, generally, and software engineering, specifically, is about much more than programming. The course unfolds in three segments. The first is a whirlwind introduction to a subset of IoT technologies. Students complete a specific task (or set of tasks) using each technology. This segment culminates in a "do-it-yourself" project, in which the students implement a simple IoT application using their basic knowledge of the technologies. The course's second segment introduces software engineering practices, again primarily via hands-on practical tutorials. In the third segment of the course, the students conceive of, design, and implement a project that uses the technologies introduced in the first segment, all while being attentive to the good software engineering practices acquired in the second segment. In addition to presenting the course curriculum, the paper also discusses a first offering of the course in a three-week summer intensive program in 2017, including assessments done to evaluate the curriculum.

1. Introduction

In recent years, computer science education has been pushed earlier and earlier; now high schoolers (even middle and elementary schoolers) are routinely exposed to programming (e.g., through Google's Hour of Code or other activities) and engineering (e.g., through robotics competitions or maker events). However, the application of good ***software engineering*** principles remains the stuff of undergraduate and graduate education. Even academic research on software engineering education remains focused on these much later stages of a student's education.

Lack of student interest in computer science education generally and software engineering specifically has received a significant amount of attention, with a particular focus on the paucity of students from traditionally underrepresented groups (e.g., women and minorities) in the field [15]. This research has demonstrated that lack of interest from students who otherwise excel in STEM (science, technology, engineering, and math) domains is due to a sense that the activities have a limited relevance [20] and a culturally inculcated "fear" that programming is inherently (too) difficult to learn [18]. However, research has also shown that exposure to hands-on computer science in the K-12 years can positively impact students' perceptions of computer science in general [11].

In this paper, we report on our experiences in taking these lessons learned about teaching computer science and applying them to teaching software engineering principles to high school students. In particular, we investigate coupling a rigorous introduction to the fundamentals of software engineering with hands-on activities utilizing the Internet of Things (IoT). Software engineering has a reputation among students as uninteresting, dry, or even "soft". The IoT, on the other hand is tactile, hands-on by nature, seen as "hard" engineering, and engaging to today's students because they can immediately relate to the applications they create.

Our concern is that early introduction of computer programming (i.e., in the K-12 years) without good software engineering practice (including a focus on requirements, design, testing, etc.) risks developing a generation of nearly capable students who are familiar with the general area of computer science but will easily become frustrated when faced with the task of building any system of real size and scale. Our hypothesis is that we can successfully couple the introduction of good software engineering practice with engaging and meaningful IoT application development activities that achieve the best of all worlds: capturing young students' interest in computing, teaching fundamental programming and engineering concepts, and introducing the importance of good software engineering practice. This paper reports on our early efforts in developing such a course.

The course is a "flipped" one [8]. Almost all the content is delivered through online modules that students consume at their own pace. Class time is devoted entirely to hands-on team activities that demonstrate software engineering principles as applied to creating IoT applications. Prior work in software engineering education has promoted the use of such flipped classrooms to deliver the Software Engineering Curriculum Model [13]. These efforts include the demonstration that a course on the fundamental elements of the software engineering process can be delivered using a flipped classroom approach [10]. However, this prior work promotes instructor-led discussions of the various phases of the software engineering process; our course instead focuses on intentional trial and error on the part of the students, followed by reflection.

The course has three major explicit goals: (1) to make basic software engineering fun and applicable, with the aim of engaging traditionally underrepresented groups in computing concepts; (2) to make young students approach problems with a design mindset, i.e., to start thinking about high-level designs before or as they start tinkering with things like breadboards and Raspberry Pis; and (3) to show students that computer science, generally, and software engineering, specifically, is about much more than programming (though programming is a substantial component).

The course encourages students to learn by (controlled) failure; learning from our failures is something of a mantra in the software engineering world [1]. The use of failure as a learning mechanism for software engineering was found to be an important element in game-based learning [22]. In our course, for instance, students are set up with tasks that are more prone to failure when good software engineering practices are not followed. Students will not be discouraged from just jumping in and trying things out; by allowing the students to fail (in a controlled way), the class then intentionally guides students in recovering from the failure in a way that is woven into the entire learning process.

2. The Course

The course unfolds in three segments. The first segment is a whirlwind introduction to a variety of IoT technologies. It is designed to allow the students to just "hack" at the different technologies. This segment does nothing to introduce any software engineering principles. Each technology is introduced as an isolated silo, with students given a specific task (or set of tasks) to complete using the technology. This segment culminates in a "do-it-yourself" project, in which, with little guidance, the students are asked to implement a simple IoT application using their basic knowledge of the related technologies. The second segment steps back from the IoT technologies to introduce principles and tools of software engineering. Periodically within this segment, these tools are explicitly related back to the do-it-yourself project or other tasks already performed. In the final segment of the course, the students are asked to conceive of, design, and implement a course project that utilizes at least three of the four technology components introduced in the first segment, all while being attentive to the good software engineering practices acquired in the second segment. In the remainder of this section, we briefly overview the curriculum from each segment. In the next section, we present some preliminary assessments used during the first offering of the course in a three-week summer intensive program offered in 2017. We also capture some of our initial insights.

2.1 Segment One: Technology Introduction

Android (Introduction to Java). Starting on the first day of the course, students are given a crash course in Java programming and asked to implement and deploy a basic Android application using a simple tutorial assignment. While the assignment launches directly into the Android framework (which is arguably unintuitive even for seasoned Java programmers [3]), the exercise is sprinkled with sidebars related to some fundamentals of object oriented programming. However, instead of coming away with a canonical "Hello World" application or a simple drawing canvas, the students have built an actual mobile application from scratch, which they deploy to an actual Android device. This task is very accessible to today's young students, meeting them where they live while demystifying that little black box in their pockets.

Philips Hue (RESTful Programming in the Web). The second technology element of the course starts with a mini-lecture on the nature of RESTful programming for the web [9], with a brief introduction to web programming more generally (e.g., HTML and HTTP). The students then perform a two-step task with a set of smart light bulbs[1]. First, the students directly issue RESTful commands to an actual light using a web interface. Second, the students augment an existing Android application that connects to and controls the lights to add random colors to the lights. For extra credit, students also add slidebars to control hue, saturation, and brightness, or they write code to sense a shaking of the phone to randomly change the light.

Introduction to Sensing (Breadboarding and the Raspberry Pi). The third technology component takes a significant step away from the traditional high-level abstractions of software engineering and delves deep into low-level sensing. Students learn about breadboarding, some simple circuits, and connecting this all to a Raspberry Pi through general purpose input/output pins. At this point, the course delves into some fundamentals of electrical engineering, with some brief lessons about circuits, resisters, capacitors, etc. The students start to see what goes into making new "things" that can be connected to their high-level application. Students create a temperature sensor, a motion sensor, and a light sensor. They experience firsthand the intricate debugging required for these hardware components. On the Raspberry Pi, students are also introduced to a second programming language (Python), where the course explicitly delivers the lesson in choosing an appropriate programming language for a given task.

Communication (Bluetooth Sockets). The final technology component introduces the students to communication via low-level sockets, specifically utilizing the Bluetooth technology available on both the Raspberry Pi and the Android device to enable information to flow between the two in both directions

[1] https://www2.meethue.com/en-us

(i.e., both from the Raspberry Pi to the Android device *and* from the Android device to the Raspberry Pi). In addition to introducing Bluetooth as a technology, students also see, for the first time, Threads and Exceptions in the Java programming language. Both are key high-level programming features. Threads are widely employed computing abstractions that allow one to enable multiple executing entities to co-exist. In this segment of the course, the students use a separate thread within an Android application to handle each communication request. Exceptions allow programs to reactively respond to abnormal conditions. In this course component, the students must specify their program's behavior in response to a communication channel breaking (e.g., because of a Bluetooth error). Again, the course tutorial for this component uses sidebars to introduce the fundamentals before the concepts are employed directly for an Android app to request and receive sensor data to be displayed on the screen. Interestingly, this technology element is by far the most daunting from an expert's perspective, but the students had no preconceptions about how hard it should be and had no more trouble with it than with the others.

Do-It-Yourself Project. Finally, to demonstrate to the students how these technologies connect, the final task in the first segment asks the students to create a circuit with an LED that is controlled by the Raspberry Pi. However, the Raspberry Pi should toggle the LED only when a user presses a button on the Android device. While all the previous segments were delivered in tutorial format (where specific code and tasks were mostly given to the students), in this mini project, the students are expected to figure out how to put the entire thing together end-to-end on their own. In addition to demonstrating that the final product "works", the students must submit a "Design document" in response to the following prompt:

We haven't learned (yet) about design documents and how they communicate the details of a design. However, let's give it our best shot anyway. Create a single page document) describing the design underlying your assignment. Think carefully about the following points:

Audience*: assume one of your classmates is your audience. You can assume a basic working knowledge of Android programming, Python programming, and connecting to the RPi through the GPIO pins.*

Functional blocks*: what are the major functional blocks and how are they connected to each other? I want you to start trying to think in abstractions instead of in individual lines of code.*

Testing: *what tests did you perform and how do they ensure correctness of your project?*

Stumbling points*: if someone were to replicate your design, what things would you recommend they watch out for?*

Resources*: were there any key resources that were really helpful to completing the assignment?*

This assignment serves as a sort of pre-test for the second segment, which, among other things, introduces good design practices. This assignment has multiple goals. First, the idea is to encourage students to think of design documentation as natural and intuitive. By asking the students to communicate their design without introducing particular techniques or diagram styles, this assignment makes the point that the goal of documenting design is to communicate on a natural level. Second, in the course of completing this design document *after* having implemented the project, the students become keenly aware of how a good, clear design can better guide the implementation phase. For instance, by thinking about the functional components of their project, students often quickly visualize alternative designs that would improve their project. By having to write down the tests they performed, students inevitably identify other tests they *should have* included. In summary, this assignment is an intentional segue into the second segment of the course.

2.2 Segment Two: Software Engineering Tools

The course's first segment takes a "get it done" kind of mentality. Students engage in practices that are deemed to be abhorrent in the software engineering community (e.g., sharing code with a collaborator via email or chat). The course's second segment highlights three of these practices and provides easy-entry tools and techniques to change them.

Version Control. The first software engineering tool the course introduces is version control. Version control systems allow software engineers to maintain a repository of artifacts related to the project, including the source code, documentation, tests, etc. The repository can be shared among collaborating developers, and it also serves as a backup of the project. Further, the history of the repository can be examined, and the current working version of the project can be "rolled back" to a previous version (e.g., to a previous version in which a major error did not occur). The course's version control module starts with a mini lecture on the importance of version control generally (both from a "backup" and history perspective and in support of collaborative projects) then uses a simple tutorial to introduce the students to both git[2] and github[3]. The course uses git because it is the most widely used version control system today and because it has a low barrier to entry. To ensure that the lesson sinks in, the students are asked to commit all their prior work (their Android projects *and* their sensor projects) to a git repository. To commit work on Android, the students use the version control tools built into the Android Studio Integrated Development Environment (IDE); to commit the sensor projects from the Raspberry Pi, the students instead use the command line from Linux. From this point forward in the class, all exchange of code between students and the instructors requires using version control (specifically, git), including submission of the remaining assignments.

[2] https://git-scm.com/

[3] https://github.com/

Software Design. This is the only course component that does not require interacting with any software or hardware introduced in the segment on software design. Instead, students are introduced to good design conceptually. The module then introduces canonical design elements from the software engineering domain (e.g., abstraction, encapsulation, coupling, cohesion, etc.) [17]. We also introduce elements of common software engineering methodologies like Agile Software Development [14] and eXtreme Programming [5] that emphasize simplicity and flexibility in design specifications. We introduce particularly use and expressive elements of the Unified Modeling Language (UML) [19], which is widely used in practice as the means to represent and communicate about software designs. As an exercise here, students are asked to revisit how they think about their do-it-yourself exercise and document how it *should* have been designed, where modules with distinct functionality are isolated one from another and interact only through well-defined interfaces. The prompt for this assignment is:

Let's revisit the Do-It-Yourself assignment where we took a crack at documenting a design without really knowing what we were doing. Briefly redo this assignment. Restructure your document (and your code!) to do (at least) the following:

Explicitly provide the requirements, architecture, technical, and user documentation.

Refactor your code to adhere to the Google Java style guide[4] (for the Android code) and Python PEP 8 style guide[5] (for the Python code).

Provide either (1) a UML-style sequence diagram showing the sequence of behaviors upon the user clicking one of the app's buttons or (2) a UML-style activity diagram showing the overall user interaction with the entire system.

Here, the students put into practice the skills they have learned conceptually, in the context of a project they implemented themselves. This post-hoc design documentation is better than none at all, but the goal is to prepare for the course's third segment, in which the students must *start* with the design documentation.

Testing. The last module in the second segment of the course introduces software testing. We start by motivating the need for rigorous testing of software through the discussion of several colossal software failures [12], and then discuss the fundamentals of testing (from black box testing [7] to white box testing [16] and why both are important; unit testing to regression testing) and discuss important concepts related to testing (e.g., test suites and coverage). To make these concepts more concrete, we then walk through specific tools for testing Android applications (the Android Testing Support Library[6], a wrapper around Junit [21] and Mockito [2]), tied to the Android applications the students have written, and integrated with the Android Studio IDE. Finally, the module introduces the notion of test-driven development [6], in which a programmer writes the tests *before* the actual implementation.

2.3 Segment Three: Course Project

Given all this preparation and directly following the segment on good software engineering practices the students are asked to conceive of, design, and execute a project of their own making in teams of two or three. At the outset of the course, we preview this project to get the students thinking about what their projects might end up being. The project should make use of the technology blocks covered in the class and encourages the students to use as many as possible.

Brainstorming. As the first step, students are encouraged to identify a *real* problem whose solution would make a difference to them or someone they know. The prompt for this phase of the project is:

Work with your partner to come up with some ideas. What kinds of things would you like to to be able to do with sensors, smart things, etc. If you have an idea that might use some devices that we don't have or haven't used yet, ask. We might have them, or we might be able to find a work-around. Come up with a few ideas. Start to draw some diagrams about the components the projects would have, and how they would fit together. Will you use Bluetooth connectivity, or would interaction through the web work? Make lists of the things that you KNOW how to do already and the unknowns.

While the students are strongly encouraged to come up with their own project ideas, we provide a small set of examples (e.g., a remote grilling meat thermometer that gradually changes the color of an interior light as the meat is more "done"; an ingress/egress sensor that counts the number of people that enter/exit a room, controlling the lights based on assumed occupancy; and a light control system that automatically adjusts a smart light based on the amount of detected ambient light. In the first iteration of the course, student projects included:

- An Android application, coupled with an LED connected to a Raspberry Pi that converted text entered in the Android application into Morse code pulses on the LED.
- A smart light application (controlled via Android) where the hue of the light reflects the sensed temperature; and another project where the hue of the light changes in response to detected motion.
- An Android application that pulled data from a weather website and adjusted the hue of a smart light based on keywords like "sunny", "cloudy", or "rainy".

[4] https://google.github.io/styleguide/javaguide.html

[5] https://www.python.org/dev/peps/pep-0008/

[6] https://developer.android.com/topic/libraries/testing-support-library/index.html

- An Android application that used a speech recognition library to allow the user to set the smart light hue by simply saying the desired color.
- An extension of our Bluetooth socket program to use the Android device as a game controller for a Simon video game that runs on the Raspberry Pi.

Design. With their problem definition in hand, the students are asked to create a design for their project. They sketch a component diagram containing the major project components and indicating how these components are connected. The design also requires determining which technologies to employ and how, based on the specific project and its requirements (e.g., should a Raspberry Pi connect directly to an Android device via Bluetooth as in the communication tutorial, or should it use a RESTful web programming API as in the web programming tutorial?). As part of the design, the students also provide at least one "user story" [17] that details how a user will interact with the completed system and at least one behavior diagram (e.g., an activity diagram or a sequence diagram). As part of this step, we also encourage the students to take a test-driven development mindset and begin conceiving of, documenting, and implementing the test cases for the project.

Implementation. The next (and most fun!) step is for the students to take this design and build the system. The team is required to use github for source control and collaboration. This forces the students to think about how to structure the source code to make it easy to maintain. While they build and code their system, students must also appropriately document what they do to ensure that they avoid entering the same pitfalls more than once. Building on the testing design in the previous phase, the students also write a testing plan, build a test suite, and ensure that their project passes the tests.

Finalizing. Finally, with the project completed, the students write their own tutorial, in the format of the tutorials used throughout the course. The goal of this tutorial is, on one hand, to document what the students did for their project. On the other hand, it should be written in such a way that everything entailed in their project can be *replicated*, another essential element of good software engineering practice.

3. Assessments

From the perspective of assessing student performance in the course, the tutorial-based modules are graded primarily based on completion. The mini-project (the LED control), the design tutorial submission, and the final project make up each student's course grade. More importantly however, are the assessments done within the class to evaluate the effectiveness of the approach to instilling software engineering principles and practices. In this section, we describe the assessments currently in place; in Section 5, we discuss our plans for future assessments.

3.1 Pre-Course Survey

Before the course begins, students complete a survey to gather details about their demographics (e.g., gender, age, etc.), preparation (e.g., type of high school, classes taken, programming experience, etc.), other factors relevant to computing as a career trajectory (e.g., interest in the course material, encouragement by others), and their preconceptions about computer science and programming (e.g., their perspective on their own abilities, their understanding of what a software engineer does, etc.). The survey is based heavily upon the Engineering and Computer Science STEM assessment tools made available by the Assessing Women and Men in Engineering Project (AWE)[7]. This survey is used in preparing the course materials to be pitched to the ability level of the class participants but also as a baseline to compare later surveys against.

3.2 Module Surveys

Upon completing each module of the course, the students are asked to complete a survey specific to that model. Each survey begins with a question to gauge the student's prior familiarity with the topic. Then each survey asks, in some form, how successful the student was at completing the assigned task(s) and how well he or she believes he or she has mastered the material. Each survey then asks (1) what was the single most difficult thing about the module; (2) what was the student's single favorite aspect of the module; and (3) what was the student's least favorite aspect of the module.

These frequent mid-course surveys achieve multiple goals. In the simplest sense, they serve as checkpoints for completion grades for the course. However, they also serve as an important mechanism for continuous improvement of the course. Finally, and perhaps essentially, they force the students to reflect about their performance and interest in the module in question. This ties directly into the course's goal of ensuring that students learn from their challenges and failures.

3.3 Post-Course Survey

For the first offering of the course, the students did a post-course survey using a generic course feedback form, which asks questions about the students' satisfaction with the course (and instructor), the amount that they learned, how interested they were in the material, and whether they would recommend the course to a friend. The students are also asked for generic free-form feedback on the course. As described in Section 5, this survey will be replaced with a more in-depth assessment that matches the pre-course survey.

4. Insights from a First Offering

The course requires a certain type of student, those with a good deal of initiative and curiosity. Further, students should expect the answers to be non-obvious and in fact a bit messy. In the first offering of the course, a subset of the students expected a lecture-based course instead of this challenging and entirely hands-on course. In the future, more clearly setting expectations early for the students may help alleviate some of these challenges. However, the students must be ready to fail and to channel that failure in a positive way; not all high school students are prepared for that.

[7] https://www.engr.psu.edu/awe/

Students moved through the course material at drastically different rates. Not all students completed the entire course, though almost all students (88% of the students in Summer 2017) completed the course project to its full specification. Because of the different rates of progress, the mechanism of self-paced tutorials was, in hindsight, essential. However, check-pointing these self-paced modules with more synchronous lectures could pull the students in the course together as a cohort better.

Finally, by the final week of the course, the students were acting as teaching assistants and tutors for one another. Some of this was instructor driven (e.g., having just shown one team how to solve a problem that another team has encountered, we would ask the first team to assist the second) and some was instead student driven (e.g., students would overhear similar problems and ask/offer help directly). In this way, the student growth in the course was significant and noticeable.

5. Looking Forward

The first offering of the course was, by many measures, very successful. The students grew tremendously not only in their capabilities related to programming and software engineering, but, more importantly, in their engagement and excitement about the field. Even so, as is always true, experience and reflection suggest several potential improvements.

During this first offering, the class met for three weeks, for two hours per week. We did not assign homework for the students to complete outside of class. Therefore, there was simply too much to learn and do during the course time. We are restructuring the course so that portions of the self-paced modules (that do not require access to the hardware in the lab) can be done outside of class, in a more truly "flipped" nature.

The relatively generic post-course survey is insufficiently matched against the tailored pre-course survey to draw end-to-end conclusions about the progression of students through this (short) course. In the future, the course will use a tailored post-course survey also based on the post-program surveys from the AWE project to see if the needle had moved on any of the survey items. getting at the real goals of the course. (Has the course changed students' perceptions of computer science and software engineering? Do students better understand design and the importance of design? Etc.)

Finally, we will also explore additional assessment techniques. For instance, to see if the course is successfully engaging students by making the material relatable, we will add Application Card [4] activities to some of the end-of-module surveys that ask the students to relate what they have just accomplished to some aspect of their everyday lives.

References

1. Tarek K. Abdel-Hamid and Stuart E. Madnick. 1990. The elusive silver lining: how we fail to learn from failure in software development.
2. Sujoy Acharya. 2014. *Mastering Unit Testing Using Mockito and JUnit*. Packt Publishing Ltd.
3. Z. Ali, J. Bolinger, M. Herold, T. Lynch, J. Ramanathan, and R. Ramnath. 2011. Teaching object-oriented software design within the context of software frameworks. *2011 Frontiers in Education Conference (FIE)*, S3G–1–S3G–5.
4. Thomas A. Angelo and K. Patricia Cross. 1993. *Classroom Assessment Techniques: A Handbook for College Teachers*. Jossey-Bass, San Francisco.
5. Kent Beck. 2000. *Extreme programming explained: embrace change*. addison-wesley professional.
6. Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.
7. Boris Beizer. 1995. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc.
8. Jacob Lowell Bishop and Matthew A. Verleger. 2013. The flipped classroom: A survey of the research. *ASEE National Conference Proceedings, Atlanta, GA*, 1–18.
9. Roy T. Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation.
10. Gerald C. Gannod, Janet E. Burge, and M. T. Helmick. 2008. Using the Inverted Classroom to Teach Software Engineering. *Proceedings of the 30th International Conference on Software Engineering*, ACM, 777–786.
11. Jung Won Hur, Carey E. Andrzejewski, and Daniela Marghitu. 2017. Girls and computer science: experiences, perceptions, and career aspirations. *Computer Science Education* 27, 2: 100–120.
12. Matt Lake. 2010. Epic failures: 11 infamous software bugs. *Computerworld*. Retrieved January 12, 2018 from https://www.computerworld.com/article/2515483/enterprise-applications/epic-failures--11-infamous-software-bugs.html.
13. Richard J. LeBlanc, Ann Sobel, Jorge L. Diaz-Herrera, and Thomas B. Hilburn. 2006. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. IEEE Computer Society.
14. Robert C. Martin. 2002. *Agile software development: principles, patterns, and practices*. Prentice Hall.
15. Phoenix Moorman and Elizabeth Johnson. 2003. Still a Stranger Here: Attitudes Among Secondary School Students Towards Computer Science. *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ACM, 193–197.
16. Thomas Ostrand. 2002. White-Box Testing. *Encyclopedia of Software Engineering*.
17. Roger S. Pressman. 2005. *Software engineering: a practitioner's approach*. Palgrave Macmillan.
18. Christine Rogerson and Elsje Scott. 2010. The fear factor: How it affects students learning to program in a tertiary environment. *Journal of Information Technology Education* 9.
19. James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified modeling language reference manual, the*. Pearson Higher Education.
20. Carsten Schulte and Maria Knobelsdorf. 2007. Attitudes Towards Computer Science-computing Experiences As a Starting Point and Barrier to Computer Science. *Proceedings of the Third International Workshop on Computing Education Research*, ACM, 27–38.
21. Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. 2010. *JUnit in Action, Second Edition*. Manning Publications Co., Greenwich, CT, USA.
22. C. G. von Wangenheim and F. Shull. 2009. To Game or Not to Game? *IEEE Software* 26, 2: 92–94.