Cite as: Boutnaru, S., & Hershkovitz, A. (2015). Software quality and security in teachers' and students' codes when learning a new programming language. *Interdisciplinary Journal of e-Skills and Life Long Learning, 11,* 123-147. Retrieved from http://www.ijello.org/Volume11/IJELLv11p123-147Boutnaru2043.pdf

Software Quality and Security in Teachers' and Students' Codes When Learning a New Programming Language

Shlomi Boutnaru and Arnon Hershkovitz
Tel Aviv University, Tel Aviv, Israel

boutnaru@mail.tau.ac.il; arnonhe@tauex.tau.ac.il

[NOTE: A short version of this paper, highlighting the main results, was presented at the 7th International Conference on Educational Data Mining (EDM 2014).]

Abstract

In recent years, schools (as well as universities) have added cyber security to their computer science curricula. This topic is still new for most of the current teachers, who would normally have a standard computer science background. Therefore the teachers are trained and then teaching their students what they have just learned. In order to explore differences in both populations' learning, we compared measures of software quality and security between high-school teachers and students. We collected 109 source files, written in Python by 18 teachers and 31 students, and engineered 32 features, based on common standards for software quality (PEP 8) and security (derived from CERT Secure Coding Standards). We use a multi-view, data-driven approach, by (a) using hierarchical clustering to bottom-up partition the population into groups based on their code-related features and (b) building a decision tree model that predicts whether a student or a teacher wrote a given code (resulting with a LOOCV kappa of 0.751). Overall, our findings suggest that the teachers' codes have a better quality than the students' – with a sub-group of the teachers, mostly males, demonstrate better coding than their peers and the students – and that the students' codes are slightly better secured than the teachers' codes (although both populations show very low security levels). The findings imply that teachers might benefit from their prior knowledge and experience, but also emphasize the lack of continuous involvement of some of the teachers with code-writing. Therefore, findings shed light on computer science teachers as lifelong learners. Findings also highlight the difference between quality and security in today's programming paradigms. Implications for these findings are discussed.

Keywords: cyber security, code metrics, software quality, software security, teachers' learning, data mining

Introduction

In its very essence, the (cognitive) constructivist theory argues that knowledge and meaning are generated from interactions between people's experiences and ideas, that is, people construct their own understanding of the world (also, of themselves) through experiencing things and reflecting upon these experiences

Editor: Gila Kurtz
Submitted: July 13, 2015; Revised: August 12, 2015; Accepted: August 16, 2015

(cf. Perkins, 1999; Phillips, 1995; Piaget, 2013). The constructivist theory has drawn huge attention in the educational theory and practice, and many pedagogical terms still in use today have been developed based on its philosophical roots, e.g., problem-based learning, situated learning, teacher as a coacher, learner-centred education (cf. Duffy & Cunningham, 1996; Richardson, 2003). Of a special interest to the current work is the development of many educational programming languages as part of the constructivism-as-pedagogy wave; the use of computer-based learning to complement Piaget's constructivism is mostly attributed to Seymour Papert (1980). Since then, "learning to program" has become by many a synonym to "learning" per se (cf. Mayer, Dyck, & Vilberg, 1986; Soloway, 1986) and "computational thinking" is often considered as key to learning (Barr & Stephenson, 2011; Gazdial, 2008; Wing, 2006). Still today, many believe that learning to code should be an integral part of any school curriculum[1]; additionally, lifelong computer training programs often include—along with more traditional modules that teach how to work with popular software (like Microsoft Word or a Web browser)—modules that teach how to code (e.g., Seals, Clanton, Agarwal, Doswell, & Thomas, 2008).

However, learning to program might be perceived as a Sisyphean task, as new programming languages are often being introduced and adopted by different communities. In other words, in practice, learning to program should not be considered as a one-time experience, but rather anyone involved with programming must be a true lifelong learner. Indeed, in recent years attention has been given to the preparation of computer science students as lifelong learners, with the presentation of various teaching methods—like problem-based learning, intentional learning environment, and collaborative learning—aimed on developing their capacity for self-direction and metacognitive awareness (Cheong, 2013; Dunlap & Grabinger, 2003). One prominent group of such lifelong programming learners are computer science teachers who often need to adapt their curriculum based on the changing popularity of programming languages. In many cases, teachers from within the system are quickly trained to be acquainted with the new curriculum. This scenario might lead to teachers teach material with which they are not fully familiar and in which they are not fully confident (Lapidot, 2004; Thompson, Bell, Andreae, & Robins, 2013).

Another concern in such cases, taking into consideration that today's young generation has easy access to vast amounts of information, is that students will progress with the new material faster than their recently-trained teachers can catch-up. This is often a result of teachers' lack of knowledge of current trends in the field and of the state-of-the-art technology, or of their difficulty to quickly adapt new technologies while their students already master them (Kordaki, 2013; Prensky, 2007).

A striking example encompassing these two issues is the adaption into schools' and universities' curricula of cyber security (also known as computer security or IT security), that is, the practice of protecting computer systems from unauthorized access, change, or destruction. This step might seem only natural, as cyber security has caught much attention of governments and industries (cf., Andrijcic & Horowitz, 2006; Cavelty, 2008; Rowe & Gallaher, 2006; Stolfo et al., 2008). Furthermore, cybersecurity may be seen as a bridge between the two classic domains of CS (software and hardware), hence requiring a new point of view on the practice of programming (Lukowiak, Radziszowski, Vallino, & Wood, 2014). In Israel, where the reported study was taken, a cyber security program has been offered in a few high-schools as part of their computer science programs. In these schools, existing computer science teachers are asked to teach the new domain which is also new to them; they are being trained in a special course that is given during after-school hours and start to teach cyber security the following school year.

[1] See, for example, the activity of the non-profit organization, Code.org, supported, among many others, by the high-tech industry giants, like Apple, Microsoft, Google, and Amazon.

The current study aims on understanding how teachers and students learn the new material from an empirical, data-driven, multi-view approach, in order to explore the differences between teachers' and students' learning. This is done by automatically extracting measures of quality and security from students' and teachers' codes (submitted during their learning) and applying data mining techniques to find patterns in them. This understanding might contribute to the pedagogy of teaching new materials, as well as to teacher development. In general, it highlights the advantages of a solid background in computer science for supporting lifelong programming learners.

Related Work

In this section we will review related work from two lines of research that are most relevant to the current study: differences between novices' and experts' programming, and code analysis (quality and security).

Novices' and Experts' Programming

Although the teachers participating in this study have not been trained before in either cyber security or in Python (one of the most popular programming language for cyber security), they may be considered as experts in the field of computer science (as well as in teaching it). The students, on the other hand, are novices in programming. Research on novices' and experts' programming knowledge had been flourished in the 1980s, with explicit comparison between these two populations and some interesting implications on teaching and learning this domain. Comparing accuracy and reaction time in solving simple text-book style programming tasks, Wiedenbeck (1985) had found that experts were significantly faster and more accurate than novices (undergraduate and graduate students) even in rather easy syntactic decisions; this finding is surprising, as the task involved consisted of merely "reading" simple code, hence supporting Wiedenbeck's automation hypothesis, according to which experts automate some simple subcomponents of the programming task. But more than automation, it was also found that experts and novices might represent differently problems they tackle; while novices begin problem representation with the problem literal features, experts first abstract an algorithm to solve it (Weiser & Shertz, 1983). This finding was later supported by Adelson (1984) and Pennington (1987), who found that experts' mental representation of programs tend to be based on procedural/abstract rather than functional/concrete units. Later, these findings were replicated by Bateson, Alexander, and Martin (1987), who found that experts did better than novices in syntactic memory (related to Wiedenbeck's automation hypothesis), strategic skills and problem solving (related to Weiser and Shertz's high-level problem representation approach).

However, as Soloway and Ehrlich (1984) found, experts' and novices' performance can be essentially the same when the program structures involved do not conform to the experts' knowledge about programming rules. Furthermore, as Adelson (1984) demonstrated, novices can perform better than experts on abstract problems once they are aided in forming an abstract representation. Without support, novices' level of abstractness of mental representation of computer programs might be poorly developed (Fix, Wiedenbeck, & Scholtz, 1993). Hence, the importance of supporting each learner with the mental model best fit to them.

The studies mentioned above used various measures to assess experts' and novices' programming skills and knowledge; these measures were mostly based on qualitative data collection (mainly programming-related tasks and interviews), rather than on assessment of the programs written by the novices/experts. Our approach is to compare between novices and experts, using automatically extracted software quality and security features. Similar approaches have been used recently for studying programming learning, mostly with a limited number of metrics, usually a-priori chosen to be mostly relevant to a given target variable (e.g., Jbara & Feitelson, 2014; Kasto &

Whalley, 2013; Piech, Sahami, Koller, Cooper, & Blikstein, 2012; Whalley & Kasto, 2014). Continuing this line of research, we take a bottom-up, rather explorative approach, relying on a comprehensive set of metrics, and assuming no relationships to any target variable.

Software Metrics

The need in defining and quantifying software quality emerged shortly after the development of the new domain of software engineering, in the late 1960s. The seminal works in this field not only defined theoretical frameworks of code quality, but also suggested explicit metrics for measuring different dimensions of it (Boehm, Brown, & Lipow, 1976; McCall, Richards, & Walters, 1976). These two frameworks share a few dimensions (e.g., correctness, reliability, and reusability), but each holds a few unique features (e.g., testability in McCall et al., 1976, and understandability in Boehm et al., 1976). In both cases, each dimension was described and detailed by means of specific metrics to be measured. Furthermore, metrics were defined with their automation in mind; as setting a numerical value for metrics might be time-consuming, subjective, and expensive, "one would prefer for large programs an automated algorithm which examine the program and produces a metric value" (Bohem et al., 1, p. 596).

Automatic evaluation of software metrics has been used in a wide variety of studies, including in the educational context. One of the most striking examples of using automated scores of code quality was suggested by Truong, Roe, and Bancroft (2005); based on automatic analysis of their programs, students were provided with immediate help, which identified correctness and logic errors and assisted them to fix these errors. Similar approaches were used to better support students' programming in real-time (e.g., Ala-Mutka, Uimonen, & Jarvinen, 2004; Vujošević-Janičić, Nikolić, Tošić, & Kuncak, 2013), as well as to auto-grade and find cheating in student assignments (e.g., Leach, 1995; Tryer, 2001; Wang, Su, Ma, Wang, & Wang, 2011). In recent years, educational data mining (EDM) and learning analytics (LA) have emerged as promising methodologies for educational research. EDM and LA are inter-disciplinary approaches to analysing data sets originated in educational contexts, using various methods and tools (cf. Siemens & Baker, 2015). These methods were incorporated into the line of research discussed here, and using these techniques, some new and exciting metrics have been added to code-analyzing, including more complex structure-based features, as well as variables measuring student-computer interaction (e.g., Berland, Martin, Benton, Smith, & Davis, 2013; Blikstein, 2011; Taherkhani & Malmi, 2013; Vihavainen, Luukkainen, & Kurhila, 2013). In this paper, we take an EDM approach, in particular applying clustering analysis and building a prediction model, based on a comprehensive set of software metrics for both quality and security.

Quality metrics

There has been a lot of research in recent years about software metrics, however with no aggregated knowledge about which metrics are more suitable to be used with which research question (cf. Kitchenham, 2010). As the main purpose of the current study was to explore the way novice students and experienced teachers learn a new topic (cyber security) – which also involves learning a new programming language (Python) – we chose to focus on software metrics derived from the standards of that language. Applying their acquired programming skills later in their working life will require current-students following some kind of style guides (for example, the ones adopted by the organization they would work for), hence the importance of this kind of measure. Therefore, sticking to coding standards/conventions of a given language is an important step within that language learning process (Pádua, 2010); this is most relevant with the modern programming languages, e.g., Java, which are more trusting and that rely on the programmer to bullet-proof her or his own code (Zaidman, 2004).

Considering the above, the Style Guide for Python Code (PEP 8) was used as the basis to the metrics involved in the current study. It is important to emphasize that keeping in mind coding style was not a goal of the courses which the participant teachers/students took, nor was it learned during these courses.

Security metrics

As mentioned earlier, the context of the current study is teachers and students who learn cyber security. Hence, it is not sufficient to consider only program quality, as security measures are mostly relevant as well. Obviously, unlike quality metrics – which have been extensively discussed for almost half a century – security metrics have been given attention only lately. But like quality metrics, there is a wide range of interpretations and meanings for the term "security metrics" (Jansen, 2010; Savola, 2009).

Consistent with the approach based on which quality metrics were chosen for this study, security measures too are based on standards of a specific programming language. However, as there are yet no such standards for Python, we referred to the commonly used *C++ Secure Coding Standard*, by Carnegie Mellon University's CERT (*SEI CERT C++ Coding Standard,* 2015). C++ seems to be a good choice, as both languages are multi-paradigm (object oriented and procedural). Furthermore, Python "is a mixture of the class mechanisms found in C++ and Modula-3" (*The Python Guide*, n.d.). Full details on both quality and security specific metrics are discussed in the following section, under Feature Engineering.

Methodology

Participants and Data

Participants in this study were 11th- and 12th-grade students (N=31) from two Israeli high-schools (one is located in a town in the outskirts of Central Israel, the other is located in the southern part of the country), 17-18 years old; and high-school computer science teachers (N=18) from different parts of Israel, 31-53 years old. The participating students were taught by two of the participating teachers.

Participant teachers attended one of two dedicated programs in cyber security, sponsored by the Israeli Ministry of Education, held between June 2012 - March 2013 and September 2013 - January 2014 (each teacher attended only one of these programs); this was done as part of their training to teach the topic the following year. The participant students took a curriculum-based cyber security program, as part of their computer science studies, during 2012/3 school year. Both of the programs had taught programming in Python. Designed in 1990, Python is an open-source, interactive, high-level programming language that supports multiple programming paradigms and offers high-level data structures (Sanner, 1999). It is free to use (even for commercial purposes) and runs across multiple platforms (including Windows, Linux, Unix, and Mac OS X). Due to its extreme power and suitability to large-scale projects, it has become the default language for Web security applications. In addition, as it is easy to learn and easy to use, Python has become one of the most popular introductory teaching languages (cf. Guo, 2014).

During these programs, the teachers and the students were assigned with programming tasks. In both cases, they started working on the tasks during class and continued it at home. All assignments had due dates. The solutions to these tasks were collected and analyzed. Overall, 109 source files were collected (68 were written by the teachers, 41 by the students), including a total of 6246 code lines. The teachers were assigned four different exercises; each focused on a different aspect of network programming using client-server architecture:

1. UDP echo server – UDP (User Datagram Protocol) is a connectionless protocol located on the "Transport Layer" of the OSI (Open Systems Interconnection) model. An echo server receives a string from a client and sends that string back to the client. In this exercise, the teachers were asked to develop an echo server using the UDP protocol.
2. Basic TCP command server – TCP (Transmission Control Protocol) is a connection-oriented protocol located on the "Transport Layer" of the OSI model. In this exercise, the teachers were asked to build a server that received TCP messages, using an application protocol that they design, and performed specific commands like sending the name of the server, sending a static number, and more.
3. Advanced TCP command server – This exercise is an enhancement of the previous one, the teachers were asked to add support for additional commands (also included the expansion of the application protocol) to the server such as retrieving date and time from the server, performing calculations on given numbers, and more.
4. Web server – A web server is a TCP based application, which accepts HTTP (Hyper Text Transfer Protocol) requests and returns relevant HTTP responses. The HTTP responses can include different content types such as HTML, pictures, video content or error messages. In the exercise, the teachers were asked to build a program and parse HTTP request. In case of a valid file request the web server sent the content of the file encapsulated in a HTTP message with valid headers.

The students were assigned three different tasks; each focused on a different aspect of network programming using client-server architecture:

1. UDP echo server – An echo server receives a string from a client and sends that string back to the client. In this exercise, the students were asked to develop an echo server using the UDP protocol.
2. Advanced TCP command server – In this exercise, the students were asked to build a server that received TCP messages, using an application protocol that they designed, and performed basic commands like sending the name of the server, sending a static number and advanced commands such as retrieving date and time from the server or performing calculations on given numbers. This exercise was a combination of both the second and the third exercises done by the teachers.
3. TCP-based Chat – A chat is a way of transferring messages between Internet users in a real-time manner. In the exercise, the students needed to design and implement a TCP based chat protocol that supported sending messages to a specific user or group and managing chat rooms.

Data is summarized in Table 1. Difference in assignments is due to the nature of the programs: While the teachers—as learners—participated in a full, planned-ahead training, the students had studied the subject as part of their regular school duties; therefore, the teachers—as instructors—were acting under the school schedule limitations, hence had time for only three tasks for their students to take.

During the pre-processing stage, we encountered tasks that were submitted by pairs or triples. In cases in which the same pair/triple had submitted all of the exercises during the program – we assigned this group's submissions to one of the group members, arbitrarily, omitting the other group members from the analysis. In cases in which pairs/triples had changed over the course of the program, we arbitrarily chose one representative for each submission and assigned this submission to her or him solely. Therefore, number of represented participants in the analysis was decreased to 17 teachers and 15 students, as shown in Table 1.

Table 1: Dataset description

Group	N	Unique exercises	Source files	Code lines
Full population				
Teachers	18 (9 males, 9 females)	4	68	3032
Students	31 (29 males, 2 females)	3	41	3214
Total	49	5	109	6246
After pairs/triples reduction				
Teachers	17 (9 males, 8 females)	4	60	2831
Students	15 (14 males, 1 female)	3	27	1878
Total	32	5	87	4709

Feature Engineering

Overall, we have engineered 32 features of three categories (details about the features are following):

- General features. A few general features were calculated for each source file, measuring, among other metrics, volume and documentation.
- Quality features. These features measure the code quality, based on the Style Guide for Python Code (PEP 8).
- Security features. In cyber security, vulnerability is a weakness that allows an attacker to perform actions not intended by the creator/owner of the application/system. As mentioned above, security-related topics were derived from CERT C++ Secure Coding Standard (which describes code-based vulnerabilities in C++). Topics related to the preprocessor were filtered out from the security standard, as Python does not have a preprocessor. Also, topics that were relevant to specific C/C++ functions not used in Python were filtered out.

Both PEP 8 and CERT's standards are widely used in code evaluation. While the programming features were automatically extracted using a standard code analysis tool, as described below (see Quality Features), the security features were extracted using scripts written by the research team. All the features are measured using a static analysis, i.e., the code is analyzed without actually executing the programs.

General features (6 Features)

For each source file, the general features are the following:

- Number of Statements, measuring the code volume;
- Number of Comment Lines;
- Documentation Rate, computed as the ratio of Number of Comment Lines to the Number of Statements;
- Number of Lines (including statements, comments and empty lines);
- File Name Length (number of characters; excluding the extension.py);
- File Name Meaningfulness –represents whether the file name hints on what the code is implementing. The values are: 1 – file name is not meaningful at all (e.g., 1.py, from which one cannot tell anything about the program implemented in this file); 2 – partly meaningful (e.g., Client.py, which does not explain to which application/protocol this

code is a client); 3 – very meaningful (e.g., ChatClientTCP.py, which means that the program implemented is a client of a chat program and that the chat protocol uses TCP as its transfer protocol). The values were given jointly by the two authors upon agreement.

Quality features (20 features)

These features were automatically extracted by running Pylint (http://pylint.org), a common source code bug and quality checker for Python which follows PEP 8 style guide. Pylint defines five categories of standard violations/errors, each of which consists of a few measures:

(1) Convention (C; 18 measures). Conventions are a set of guidelines that recommend programming style, practices, and methods for each aspect of a program. These guidelines are mainly focused on software structural quality. Convention measures indicate a standard violation, for example, when the name of an attribute/class/function/variable does not match a regular expression defined in the standard;
(2) Warning (W; 61 measures). This type refers to Python-specific problems, that is, not matching Python's best practices. Such problems may cause bugs in run time. For example, the existence of unused import from wildcard import;
(3) Error (E; 32 measures). This indicates probable bugs in the code that relate to general programming concepts. For example, the use of a local variable before its assignment;
(4) Refactor (R; 15 measures). This means a "bad smell" code. Code refactoring (also known as decomposition) is the process of restructuring existing computer code without changing its external behaviour, mainly for reducing its complexity, easing its readability, and improving its maintainability. An example to this violation might be a function or a method which takes too many variables as input;
(5) Fatal. This is triggered if an error occurred which prevented Pylint from processing the code. Since fatal messages represent errors in Pylint processing and not in the source file itself, we excluded them from this study.

Pylint scans the code and returns a list of measures for which violations/errors were found, along with their count (a 0-value was considered for non-triggered measures). Based on Pylint output, the following features were computed under each category:

- Mean Count (C/W/E/R) – for each category, count of violations/errors was averaged across all this category's measures.
- Normalized Mean Count (C/W/E/R) – for each category, Mean Count divided by the code size (Number of Statements);
- Rate of Triggered Measures (C/W/E/R) – for each category, number of triggered measures divided by total number of measures in this category;
- Triggered Category (C/W/E/R) – for each category, this feature indicates whether at least one measure of it was triggered.
- Normalized Triggered Category (C/W/E/R) – Triggered Category, normalized by the code size (Number of Statements).

Security features (6 features)

As mentioned above, security features are derived from CERT C++ Secure Coding Standard. The following are the features extracted, all are binary, indicating whether the relevant mechanism was implemented (1) or not (0). An implemented mechanism (i.e., a 1-value) is an indication of the programmer's attempt to protect the code from potential security vulnerabilities; pay attention that the last feature, Client-Side-Only Security, refers to a bad practice (security check should always be done at least on the server-side), hence its meaning is reversed and for it a 1-value denotes a possible security vulnerability.

- Input Validation (the process of ensuring that a program operates on clean, correct and expected input);
- Anti-Spoofing Mechanism (spoofing attack is a situation in which an attacker masquerades as another entity by sending specially crafted data that seems as it was send from the legitimate source);
- Bound Checking (any method of detecting whether a variable is within some range before it is used. It is usually used to ensure that a number fits into a given data-type, or that an array index is within the bounds of a given array);
- Checking for Errors (not checking return codes for errors can cause logical security bugs/crashing of the program that can cause Denial of Service attacks).
- Sensitive Data Encryption;
- Client-Side-Only Security (a scenario in which the server relies on protection mechanisms placed on the client side only. In such cases, an attacker can modify the client-side behavior to bypass the security mechanisms, which can lead to unauthorized, not intended interactions between the client and the server).

For the participant-level analysis (descriptive statistics, hierarchical tree), each feature was averaged across each participant's source files.

Results

We first present descriptive statistics of the software metrics (general, quality, and security). Then, we present the results of the hierarchical clustering that was run on the full, combined population, in order to examine a data-driven partition of it based on the metrics. Finally, we build a prediction model at the code-level, for examining whether the software metrics can differentiate between teachers' and students' codes.

Descriptive Statistics

As described above, 60 source files of 17 teachers and 27 source files of 15 students (after reducing due to submissions in pairs/triples) were analyzed, and 32 software metrics were computed for each source file. Then, values of the different metrics were averaged for each participant across all of her or his source files (analysis at the code-level will be presented in Teacher/Student Classification). We will now present a comparison between students' and teachers' feature values.

General features

Means (and standard deviations) of the general features for teachers and students are summarized in Table 2, along with two-tailed t-test results of comparing between the two groups. Differences in means of four (out of six) general metrics are significant: Number of Statements ($p<0.05$), Number of Lines ($p<0.05$), File Name Length ($p<0.01$), and File Name Meaningfulness ($p<0.01$); on average, students' programs were longer than the teachers', and teachers' file names were longer and more meaningful than the students'. The difference regarding code size (Number of Statements and Number of Lines) might indicate the teachers have a better grasp of the concept of programming with Pyhton, as "Python is a very expressive language, which means that we can usually write far fewer lines of Python code than would be required for an equivalent application written in, say, C++ or Java" (Summerfield, 2010, p. 1).

Regarding file names' length and meaningfulness – teachers demonstrate a better practice of file naming, which might be a result of their experience in programming and/or teaching computer science classes. No significant differences were found between the means of the two documenta-

tion-related features. Average Documentation Rate was 0.1, a reasonable documenting practice in Python.

Table 2: Descriptive statistics, two-tailed t-test results for general features (one decimal place representation unless mean<0.1); grey-shaded rows have significant difference

Variable	Mean (SD) N=32	Mean (SD), Teachers, N=17	Mean (SD), Students, N=15	t(30)[a]
Number of Statements	51 (28.3)	40.5 (19.7)	62.9 (32.3)	2.3*, df=22.6[b]
Number of Comments	6.1 (7.4)	5.5 (7.8)	6.8 (7.0)	0.5
Documentation Rate	0.1 (0.1)	0.1 (0.2)	0.1 (0.1)	-0.4
Number of Lines	56.9 (29.6)	45.4 (23.8)	69.7 (30.9)	2.5*
Name Length	10.8 (5.1)	12.9 (4.0)	8.4 (5.2)	-2.8**
Name Meaning	1.3 (0.5)	1.6 (0.4)	0.9 (0.5)	-4.3**

* $p<0.05$, ** $p<0.01$. [a] Unless otherwise stated, df=30.

[b] Levene's test for equality of variance was significant, hence equal variances not assumed.

Quality features

Means (and standard deviations) of the software quality features for teachers and students are summarized in Table 3, along with two-tailed t-test results of comparing between the two groups.

Means of eight (out of twenty) quality metrics are significantly different between groups, all are of the convention and warning categories: Mean Count C, Mean Count W, Normalized Mean Count C, Normalized Mean Count W, Rate of Triggered Measures C, Rate of Triggered Measures W, Trigged Category W, and Normalized Triggered Category C (all at $p<0.01$, except Triggered Category W, at $p<0.05$). On average, the students had more convention- and warning-type violations than the teachers (with only one exception, Normalized Triggered Category, which will be addressed immediately). As convention guidelines help improve code readability and make software maintenance easier, these differences might indicate that the teachers, based on their experience, migrate rather smoothly to programming in a new language.

Pay attention to the opposite direction difference between students and teachers in Normalized Triggered Category C. This is a direct result of Triggered Category C getting a value of 1 for both students and teachers and of Number of Statements being larger for students that it is for teachers (as presented in General Features); the variable Normalized Triggered Category C is a ratio of these two variables.

Table 3: Descriptive statistics, two-tailed t-test results for quality features (one decimal place representation unless mean<0.1 or difference needs to be shown); grey-shaded rows have significant difference

Variable	Mean (SD) N=32	Mean (SD), Teachers N=17	Mean (SD), Students N=15	t(30)[a]
Mean Count C	72.3 (56.7)	40.8 (28.8)	108.0 (60.0)	4.0**, df=19.6[b]
Mean Count W	56.9 (70.9)	20.7 (37.6)	97.8 (78.4)	3.5**, df=19.5[b]
Mean Count E	1.4 (1.5)	1.3 (1.0)	1.5 (2.0)	0.5, df=19.5[b]
Mean Count R	0.2 (0.3)	0.1 (0.4)	0.2 (0.3)	0.6
Normalized Mean Count C	0.11 (0.04)	0.09 (0.04)	0.13 (0.03)	3.6**, df=26.7[b]
Normalized Mean Count W	0.02 (0.03)	0.01 (0.02)	0.04 (0.03)	2.9**, df=21.6[b]
Normalized Mean Count E	– [c]	– [c]	– [c]	0.05
Normalized Mean Count R	– [c]	– [c]	– [c]	1.1
Rate of Triggered Measures C	0.4 (0.1)	0.3 (0.1)	0.4 (0.1)	4.5**
Rate of Triggered Measures W	0.05 (0.04)	0.03 (0.03)	0.07 (0.04)	3.5**
Rate of Triggered Measures E	0.01 (0.01)	0.01 (0.01)	0.01 (0.01)	-0.1, df=21.6[b]
Rate of Triggered Measures R	0.01 (0.02)	0.01 (0.02)	0.01 (0.01)	0.7
Triggered Category C	1 (0)	1 (0)	1 (0)	N/A
Triggered Category W	0.7 (0.4)	0.6 (0.4)	0.9 (0.3)	2.7*, df=28.1[b]
Triggered Category E	0.4 (0.3)	0.4 (0.3)	0.4 (0.4)	-0.6, df=23.8[b]
Triggered Category R	0.2 (0.3)	0.1 (0.3)	0.2 (0.3)	1.2
Normalized Triggered Category C	0.03 (0.02)	0.04 (0.02)	0.02 (0.01)	-3.1**
Normalized Triggered Category W	0.02 (0.01)	0.02 (0.02)	0.02 (0.01)	0.6
Normalized Triggered Category E	0.01 (0.01)	0.01 (0.01)	0.01 (0.01)	-1.1
Normalized Triggered Category R	– [c]	– [c]	– [c]	0.2

* $p<0.05$, ** $p<0.01$. [a] Unless otherwise stated, df=30.

[b] Levene's test for equality of variance was significant, hence equal variances not assumed.

[c] Mean value was smaller than 0.01.

Security features

Overall, both teachers and students showed low levels of implementing security mechanisms. Results are summarized in Table 4. Both teachers and students implemented none of Anti-Spoofing Mechanisms and Sensitive Data Encryption. As for Input Validation and Checking for Errors – on average, students implemented more mechanisms than teachers regarding these features ($p<0.05$). This might be a result of the teachers, learning from their own fresh experience, emphasizing these subjects to their students.

As for Client-Side-Only Security, recall that a 0-value for this feature denotes a proper security implementation. As seen from Table 2, mean value for this feature was 0 for the teachers; however, as the teachers had barely implemented any security mechanism, this value cannot be inter-

preted as a good security practice. The students, with relatively a high mean value of Client-Side-Only Security (0.5, significantly different from the teachers' mean, at $p<0.01$), demonstrate poor security design that is focused mostly at the client-side.

Table 4: Descriptive statistics, two-tailed t-test results for security features (one decimal place representation unless mean<0.1); grey-shaded rows have significant difference

Variable	Mean (SD) N=32	Mean (SD), Teach. N=17	Mean (SD), Stud. N=15	t[a]
Input Validation	0.06 (0.17)	0 (0)	0.13 (0.23)	2.3*, df=14.0
Anti-Spoofing Mechanism[b]	0 (0)	0 (0)	0 (0)	N/A
Bound Checking	0.10 (0.20)	0.04 (0.12)	0.17 (0.25)	1.9, df=20.1
Checking for Errors	0.18 (0.35)	0.04 (0.12)	0.33 (0.45)	2.5*, df=15.8
Sensitive Data Encryption	0 (0)	0 (0)	0 (0)	N/A
Client-Side-Only Security[c]	0.21 (0.42)	0 (0)	0.5 (0.52)	3.3**, df=11.0

* $p<0.05$, ** $p<0.01$.

[a] Levene's test for equality of variance was significant, hence equal variances not assumed.

[b] For this case, for students, N=12. [c] For this case, for teachers, N=16, and for the students, N=12.

Hierarchical Clustering of the Participants

In order to explore similarities between the participants, with regards to their code metrics, we used a hierarchical cluster analysis. This method makes a partition of the population into groups where subjects in each group are similar (or closer) to each other than subjects in other groups, by terms of a given metrics; this is a bottom-up, unsupervised method that makes no prior assumptions on the way the data is categorized (cf. Kaufman & Rousseeuw, 2009). In recent years, and mainly since the emergence of the educational data mining (EDM) and learning analytics (LA) approaches to educational data, cluster analysis has been extensively used in educational research (cf. Antonenko, Toy, & Niederhauser, 2012; Baker & Siemens, 2014; Peña-Ayala, 2014; Romero & Ventura, 2010). As our purpose was to discover hidden patterns in the data, no assuming on any a-priori partition of the population, we find this approach most suitable to our needs; the metrics by which the population is partitioned is based on the calculated metrics. We used Ward's method (Ward, 1963) for clustering by Pearson correlation. Features were standardized using Z-scores before clustering. Analysis was computed using SPSS 18. A dendrogram representing the clustering process is presented in Figure 1. The vertical lines determine which participants were grouped together and at which stage of the algorithm (from left to right). As can be clearly seen, the process resulted with a division to two clusters, holding 9 and 23 participants. The small cluster holds teachers only; the large one holds all the students (15) and 8 additional teachers. Examining mean feature values between the two clusters, and comparing these with the descriptive statistics presented in previous section, raises some interesting insights.

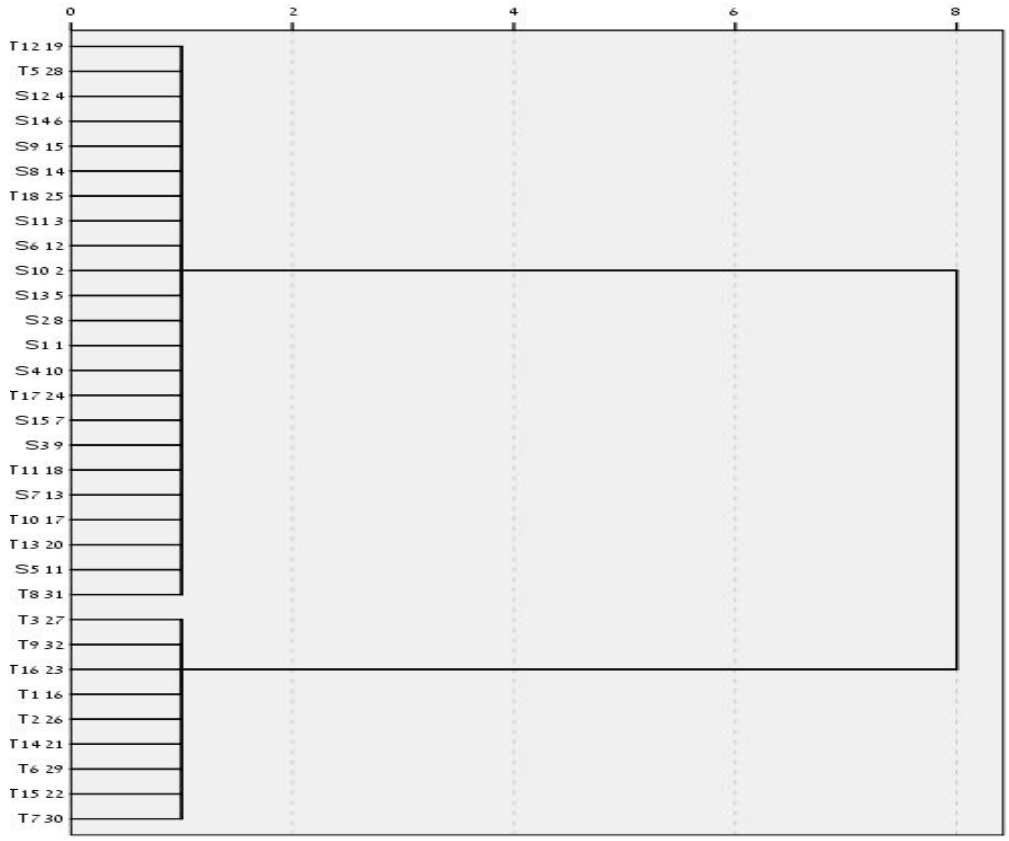

Figure 1: Dendrogram of the hierarchical clustering process. X-axis represents distance between clusters

First, we will examine mean differences between the clusters and will compare these results with mean differences between students and teachers (presented in the previous section). These comparisons are summarized in Table 5, where each variable is noted for significance/insignificance difference when its mean is compared between clusters and between students/teachers. Full results of mean comparison between the two clusters are presented in Tables 6-8. We will now focus only on these variables the means of which were found to be significantly different when comparing between cluster and were not significantly different when comparing between teachers/students, and vice versa.

Table 5: Significance ($p<0.05$, marked with +) or insignificance (-) mean difference in two-tailed t-tests regarding the research variables

Variable	Comparing Students, Teachers	Comparing Clusters
General Features		
Number of Statements	+	-
Number of Comments	-	-
Documentation Rate	-	-
Number of Lines	+	-
Name Length	+	+
Name Meaning	+	+
Quality Features		
Mean Count C/W	+	+
Mean Count E/R	-	-
Normalized Mean Count C/W	+	+
Normalized Mean Count E/R	-	-
Rate of Triggered Measures C/W	+	+
Rate of Triggered Measures E	-	-
Rate of Triggered Measures R	-	+
Triggered Category C	N/A	N/A
Triggered Category W	+	+
Triggered Category E	-	-
Triggered Category R	-	+
Normalized Triggered Category C	+	-
Normalized Triggered Category W/E/R	-	-
Security Features		
Input Validation	+	+
Anti-Spoofing Mechanism	N/A	N/A
Bound Checking	-	+
Checking for Errors	+	+
Sensitive Data Encryption	N/A	N/A
Client-Side-Only Security	+	+

Significant Difference between Clusters, not between Teachers/Students

Interestingly, means of two of the Refactor features are significantly different when comparing between clusters, both refer to the overall triggering of measures under this category: a) Rate of Triggered Measures R, with $t(df=20.5)=2.2$, at $p<0.05$; and b) Triggered Category R, with $t(df=24.4)=2.2$, at $p<0.05$. Levene's test for equality of variance resulted with a significant result in both cases, hence equal variances were not assumed. Recall that none of the Refactor features was significantly different between teachers and students. In both cases, the means in the teachers-only cluster were lower than the means in the mostly-students cluster (i.e., the teachers had demonstrated better security design). As Refactor measures refer to a "bad smell" code, the differences in this category might indicate that the teachers in the teacher-only cluster are more experienced in regulating their own programming and recognizing seemingly-suspicious code than the students and the other teachers.

Bound Checking was also found significantly different between the two clusters, with $t(df=19.1)=2.2$, at $p<0.05$ (again, equal variances were not assumed as for Levene's test significant result). Here, mean value for the teachers-only cluster is lower than the mostly-students cluster, indicating poorer security design of members of the former comparing to the latter. This is probably because, as was shown in the previous section, the Bound Checking security mechanism was mostly implemented by the students.

Overall, these differences highlight that the teacher-only clusters hold an important subset of the teachers, the member of which differ not only from the students, but also from their other peers. The data-driven partition taken here made it possible to distinguish between the two groups based not on their role in class, but rather on their actual code-writing measures, hence finding some hidden patters.

Significant Differences between Teachers/Students That Became Non-significant

Means of two of the General features previously found significantly different between teachers/students are now non-significant when comparing between clusters: *Number of Lines*, *Number of Statements*, and *Normalized Triggered Category C*. As Normalized Triggered Category C is the ratio of Triggered Category C – for which all of the participants got a value of 1 – to Number of Statements, and as Number of Statements and Number of Lines are highly correlated– with Pearson's $r=0.983$, at $p<0.01$ – these three features are related. Hence, the teachers-only cluster and the mostly-students cluster do not differ from each other with regards to their general code characteristics.

The cluster analysis resulted with a clear distinction between two groups of learners. Specifically, it found a sub-group of the teachers that is different from the rest of the teachers – and along with that from all the students – mostly in its quality features (convention-, warning-, and refactor-related). This subgroup's members produce code that is higher in its quality measures compared to the other participants; this might indicate that teachers in the teacher-only cluster facilitate their learning of new material differently than the rest of the participants. To this end, gender issues might be relevant, as the teachers-only cluster holds 2 female teachers and 7 male teachers, while the mostly-students cluster holds 2 male teachers and 6 female teachers (as mentioned before, almost all participant students were males, making any gender-wise comparison impossible). This brings to mind an interesting recent finding about gender differences in adopting computer curricular changes. Investigating a rapid change in CS curriculum in New Zealand, Thompson et al. (2013) found that only 73% of the female teachers intended to use the new programming and CS standards presented in the new curriculum, compared to 91% of the male teachers.

Table 6: Mean comparison (two-tailed t-test) for the general features among the two clusters (one decimal place representation unless mean<0.1); grey-shaded rows have significant difference

Variable	Mean (SD), Mostly-students Cluster, N=23	Mean (SD), Teachers-only Cluster, N=9	t(30)[a]
Number of Statements	56.0 (30.6)	38.3 (16.0)	-0.8
Number of Comments	6.1 (7.6)	6.2 (6.2)	-0.04
Documentation Rate	0.08 (0.07)	0.2 (0.2)	-1.4, df=8.9[b]
Number of Lines	62.2 (31.1)	43.1 (20.7)	1.7
Name Length	9.0 (4.6)	15.2 (3.2)	-3.7**
Name Meaning	1.1 (0.5)	1.7 (0.4)	-2.9*

* p<0.05, ** p<0.01. [a] Unless otherwise stated, df=30.

[b] Levene's test for equality of variance was significant, hence equal variances not assumed.

Table 7: Mean comparison (two-tailed t-test) for the quality features among the two clusters (one decimal place representation unless mean<0.1); grey-shaded rows have significant difference

Variable	Mean (SD), Mostly-students Cluster, N=23	Mean (SD), Teachers-only Cluster, N=9	t(30)[a]
Mean Count C	90.7 (56.7)	25.5 (13.4)	5.2**, df=27.2
Mean Count W	75.9 (75.4)	8.2 (13.3)	4.1**, df=25.2
Mean Count E	1.3 (1.7)	1.6 (1.0)	-0.4
Mean Count R	0.2 (0.4)	0.05 (0.1)	1.8, df=27.9
Normalized Mean Count C	0.1 (0.03)	0.06 (0.03)	6.1**
Normalized Mean Count W	0.03 (0.04)	0.003 (0.004)	3.8*, df=23.4
Normalized Mean Count E	0.0 (0.001)	0.001 (0.001)	-0.8
Normalized Mean Count R	0.0[c] (0.0[c])	0.0[c] (0.0[c])	2.0, df=26.7
Rate of Triggered Measures C	0.4 (0.08)	0.3 (0.05)	4.1**
Rate of Triggered Measures W	0.07 (0.04)	0.02 (0.02)	3.3*
Rate of Triggered Measures E	0.01 (0.01)	0.02 (0.003)	-0.8, df=27.3
Rate of Triggered Measures R	0.01 (0.02)	0.002 (0.004)	2.5*, df=27.2
Triggered Category C	1.0 (0.0)	1.0 (0.0)	N/A
Triggered Category W	0.8 (0.4)	0.5 (0.4)	2.4*
Triggered Category E	0.4 (0.4)	0.5 (0.1)	-1.0, df=29.8
Triggered Category R	0.2 (0.3)	0.05 (0.1)	2.5*, df=29.8
Normalized Triggered Category C	0.03 (0.02)	0.04 (0.01)	-1.2
Normalized Triggered Category W	0.02 (0.01)	0.02 (0.02)	0.2
Normalized Triggered Category E	0.008 (0.01)	0.01 (0.01)	-1.1
Normalized Triggered Category R	0.003 (0.01)	0.0[c] (0.0)	1.3

* p<0.05, ** p<0.01. [a] Unless otherwise stated, df=30.

[b] Levene's test for equality of variance resulted with a significant result, hence equal variances not assumed.

[c] Value was smaller than 0.01.

Table 8: Mean comparison (two-tailed t-test) for the security features among the two clusters (one decimal place representation unless mean<0.1); grey-shaded rows have significant difference

Variable	Mean (SD), Mostly-students Cluster, N=23	Mean (SD), Teachers-only Cluster, N=9	t[a]
Input Validation	0.09 (0.2)	0.0 (0.0)	2.2*, df=22
Anti-Spoofing Mechanism[b]	0.0 (0.0)	0.0 (0.0)	N/A
Bound Checking	0.1 (0.2)	0.01 (0.04)	2.4*, df=25.5
Checking for Errors	0.2 (0.4)	0.0 (0.0)	3.0*, df=22
Sensitive Data Encryption	0.0 (0.0)	0.0 (0.0)	N/A
Client-Side-Only Security[c]	0.3 (0.5)	0.0 (0.0)	2.9*, df=18

* $p<0.05$, ** $p<0.01$. [a] Levene's test for equality of variance resulted with a significant result, hence equal variances not assumed.

[b] For this case, for the mostly-students cluster, N=20.

[c] For this case, for the mostly-students cluster, N=19.

Code Writer Prediction Model

As a final step, and in order to highlight the findings from yet another angle, we tried to build a classifier, at the code-level, that will predict whether a program was submitted by either a student or a teacher. In this case, the analysis is supervised, as we know ahead which code was written by a student and which was written by a teacher, and our goal is to predict it based on the code metrics. That is, we would like to build a classifier (a binary predictor) that, based on a code's features, will predict who wrote this code. For doing this, we chose to use a decision tree model, mostly because of it being simple and easily interpretable (cf. Quinlan, 1996); decision trees have been extensively used in the educational research in recent years (cf. Baker & Siemens, 2014; Peña-Ayala, 2014; Romero & Ventura, 2010).

As detailed in Participants and Data section, 87 code source files were used. This set of codes, along with their features (at this stage, back to code-level calculation of features), had served for the prediction model building. We developed a decision tree model, using RapidMiner 5.3 (Mierswa, Wurst, Klinkenberg, Scholz, & Euler, 2006), with its default parameters. Forward feature selection was implemented manually. Performance was tested using kappa and was validated using leave-one-out cross-validation.

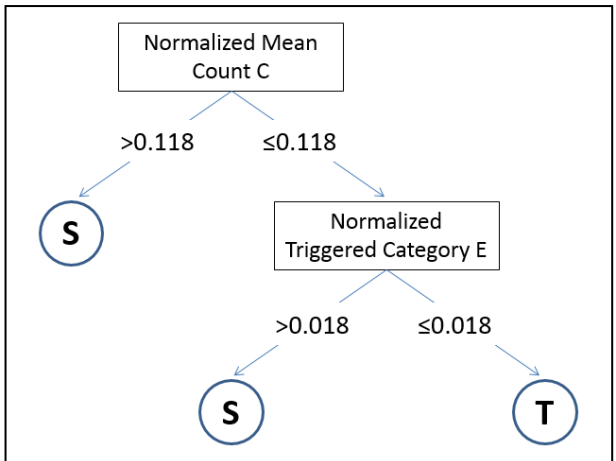

Figure 2: Best prediction model (S=Student, T=Teacher)

The best model found, presented in Figure 2, is relatively simple, having only two features – Normalized Mean Count C, and Normalized Triggered Category E – three

leaves and a total height of two. This model has a very good kappa value of 0.751, indicating on its high level of generalizability.

Based on this tree, a code with Normalized Mean Count C>0.118 is predicted to be written by a student. A code with a lower value of Normalized Mean Count C is then tested for Normalized Triggered Category E; higher values (>0.018) are predicted to be written by students, lower values (≤0.018) are predicted to be written by teachers. This model highlights the already known difference in convention violations between teachers and students. However, it also highlights an interaction between the convention feature and a feature that refers to errors. Error-related features did not show significance earlier. This interesting result suggests that students and teachers that are relatively good in convention-keeping, might still pay attention differently to probable bugs. Here again, teachers might use their experience in programming (or in teaching programming) for thinking about different scenarios and extreme cases that might lead to bugs.

It is important to emphasize that security-related features, as well as general features, are absent from the model, highlighting the fact that students and teachers are mostly differed by the quality of their code. The non-differentiability of the general features might indicate that the submitted codes are straightforward solutions to relatively simple tasks; the non-differentiability of the security features is probably related to the very low levels of security mechanisms implemented by both students and teachers, as was previously shown; therefore, the only inherent meaningful difference at the code-level remains in the quality features, indicating the main advantage experienced teachers have over students when it comes to studying a new programming language.

Discussion

In this study, we analyzed students' and teachers' computer programs for both quality and security; the programs were given as exercises in a cyber security training, a topic new for both populations. Software quality measures indicate that both groups have violations/errors, mainly in the Python-specific features (convention, warning), which demonstrate the difficulty of learning a new programming language. Students' (obviously novices) difficulties are expected; teachers' difficulties may be looked at in two opposite ways: on the one hand, teachers did better than students in all software quality metrics, demonstrating their experience and expertise; on the other hand, the very existence of these violations/errors may hint that the teachers had struggled with the new material just like novices do. In this light, our findings support Liberman, Ben-David Kolikant, and Beeri's (2012) preliminary results about the "regressed experts", stating that in such scenario, the teachers practice some elements of novices but use their experience as a leverage to improve knowledge in the new content. Recall that about half of the teachers were clustered together with the students when defining similarity based on the software metrics (using hierarchical clustering). This group of teachers, the members of which proved a slower adaptation to the new curriculum, might demonstrate the problem in the training of secondary school computer science (CS) teachers. As Ginat (2000) put it, "many teachers remained with a rather narrow perception of CS. In particular, many still perceive the teaching of introductory CS as tutoring of language syntax and technicalities, and fail to recognize the importance of design and analysis consideration." While Liberman, Ben-David Kolikant, and Beeri's description relates to our surface-level findings (based on descriptive statistics), shedding light on the more severe problem presented by Ginat was enabled by using a bottom-up analysis approach (based on data mining). This demonstrates the advantage of our multi-view approach. Additionally, the prediction model highlighted some further fine-grained differences between students and teachers, emphasizing the role of quality-related features, in which teachers most benefit from their experience. Repeating the current study in a larger population is recommended in order to empirically measure the extent of these two phenomena, both of which are strongly related to the ability of CS teachers to be trained in new programming languages, as required in today's ever-changing curriculum era.

Viewed from a broader perspective, these findings shed light on CS education as a lifelong learning experience. The rapid technological developments over recent years make CS an ever-changing field, and as a result, people engaged with it are required to be lifelong learners (Fischer, 2000; Guzdial & Weingartn, 1995). Among these are CS teachers who need to be constantly updated in order to teach their students the most relevant content. Sometimes, the need to get updated is forced on these teachers as a result of curricular changes. Indeed, lifelong learning has been noted as an integral part of CS teacher preparation, side by side with expertise in knowledge and in teaching strategies (East, Bentley, Kmoch, Rainwater, & Stephenson, 2011). As we showed in this study, it might be advisable to emphasize the need for lifelong learning in CS teacher training. This need should also be accompanied by research focusing on the most efficient ways for CS teachers to keep updated in content-, technology-, and pedagogy-related topics.

On top of these two explanations to CS teachers' training in new material, gender issues might explain some differences too, as was pointed out in our findings and was recently found by Thompson et al. (2013). As Paechter (2003) suggested, teachers' reactions to curriculum change are strongly related to teacher identities, in specific gender: while female teachers see themselves as teachers first, content experts second, for male teachers the order of importance is reversed. This observation might dramatically affect teachers' attitude towards curricular changes. In addition to demonstrating different attitudes towards changes in the curriculum, compared with their male peers (which might, in turn, affect their learning of new material), female CS teachers might also learn the new content differently than their male colleagues, due to previously observed gender differences in CS learning (e.g., Beyer, Rynes, Perrault, Hay, & Haller, 2003; Murphy et al., 2006; Vilner & Zur, 2006). Generally, our finding regarding gender differences is in line with many other studies that have pointed out to gender-differences in computer science education (cf. Hayes, 2008; Mihalcea & Welch, 2015; Webb & Miller, 2015). These differences should be taken into consideration by policy makers and teacher development program leaders, in order to ease the processes of female CS teachers adapting to curricular changes.

Meanwhile, a simple solution might assist both students and teachers to learn in the CS with programming tasks: measuring software quality and security metrics in real-time (i.e., while writing the code), hence enabling contextual feedback that might result in a better code and promote better learning (cf. Ala-Mutka, 2007; Ng, Vee, Meyer, & Mannock, 2006; Truong et al., 2005; Wang et al., 2011). Popular IDEs (Integrated Development Environments) already provide integration with tools like Pylint (e.g., Emcas, VIM, Eclipse, Komodo, WingIDE, gedit, and pyscripter), so using such software might ease the measuring task, as was indeed reported in Robles and González-Barahona (2013). This idea is in line with a strand of research that suggested supporting novice programming learners by adding components during program writing, running or debugging (e.g., Brusilovsky, 1994, Egan & McDonald, 2014; Hundhausen & Brown, 2007; Kiesmuller, 2009). Hence, this is another important direction for further research that might eventually assist people to acquire a new programming language more efficiently.

As our results suggest, codes with higher software quality are not necessarily better secured. Overall, codes of the participant teachers were of higher quality comparing to the students' codes; however with regards to the measurable security features, the opposite was observed. In recent years, much attention has been put on exploring the relations between software quality metrics and software vulnerability; most of these studies have found that the former can predict the latter (e.g., Chowdhurry & Zulkernine, 2011; Moshtari, Sami, & Azimi, 2013; Shin & Williams, 2013). However, most of these studies were analysing big (in terms of code size) commercial software (e.g., Mozilla Firefox, Linux Kernel, Eclipse, etc.) that are usually written collaboratively by hundreds, sometimes thousands, of people. Hence, their findings are barely interpretable at the programmer level. The current study sheds light on the individual code-writer, so we can conclude from it about educating software engineering students and lifelong programming learners.

From what we have found, educating in secure programming might probably be done in parallel to teaching programming practices.

Generally, due to the fact that most of the features (quality features and cyber security features) shown in the study are relevant to other programming languages (such as C/C++/C#/Java/Ruby), our findings might generalize to other environments, and we plan on running similar, extended studies in other programming scenarios.

References

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of experimental Psychology, 10*(3), 483-495.

Ala-Mutka, K. (2007). A survey of automated assessment approaches for programming assignments. *Computer Science Education, 15*(2), 83-102.

Ala-Mutka, K., Uimonen, T., & Järvinen, H.M. (2004). Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education, 3*, 245-262. Retrieved from http://www.jite.org/documents/Vol3/v3p245-262-135.pdf

Andrijcic, E. & Horowitz, B. (2006). A macro-economic framework for evaluation of cyber security risks related to protection of intellectual property. *Risk Analysis, 26*(4), 907-923.

Antonenko, P. D., Toy, S., & Niederhauser, D. S. (2012). Using cluster analysis for data mining in educational technology research. *Educational Technology Research and Development, 60*(3), 383-398.

Baker, R. & Siemens, G. (2014). Educational data mining and learning analytics. In K. Sawyer (Ed.), *Cambridge handbook of the learning sciences* (2nd ed.) (pp. 253-274).

Barr, V. & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads, 2*(1), 48-54.

Bateson, A. G., Alexander, R. A., & Murphy, M. D. (1987). Cognitive processing differences between novice and expert computer programmers. *International Journal of Man-Machine Studies, 26*(6), 649-660.

Berland, M., Martin, T., Benton, T., Smith, C. P., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. *The Journal of the Learning Sciences, 22*(4), 564-599.

Beyer, S., Rynes, K., Perrault, J., Hay, K., & Haller, S. (2003). Gender differences in computer science students. *Proceedings of SIGCSE '13 (Reno, NV, February 19-23)*, 49-53.

Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. *In Proceedings of the 1st International Conference on Learning Analytics and Knowledge (Banff, AB)*, 110-116.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *In Proceedings of the 2nd International Conference on Software Engineering (San Francisco, CA)*, 592-605.

Brusilovsky, P. (1994). Explanatory visualization in an educational programming environment: Connecting examples with general knowledge. *In Proceedings of the 4th International conference EWHCI'94 (St.Petersburg, Russia, August 2-6)*, 202-212.

Cavelty, M. D. (2008). *Cyber-security and threat politics: US efforts to secure the information age*. New York, NY: Routledge.

Cheong, L.H. (2013). A problem-based learning approach to teaching a computer programming language. *International Proceedings of Economics Development and Research, 66*, 68-73.

Chowdhury, I., & Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture, 57*(3), 294-313.

Duffy, T. M., & Cunningham, D. J. (1996). Constructivism: Implications for the design and delivery of instruction. In D. Jonassen (Ed.), *Handbook of research for educational communications and technology* (pp. 170-198). New York, NY: Simon & Schuster Macmillan.

Dunlap, J. C. & Grabinger, S. (2003). Preparing students for lifelong learning: A review of instructional features and teaching methodologies. *Performance Improvement Quarterly, 16*(2), 6-25.

East, J. P., Bentley, C., Kmoch, J., Rainwater, S., & Stephenson, C. (2011). NCATE standards for preparation of secondary computer science teachers. In *Proceedings of SIGCSE 2011 – The 42st ACM Technical Symposium on Computer Science Education (pp. 243-244)*. New York, NY: ACM New York.

Egan, M. H., & McDonald, C. (2014). Program visualization and explanation for novice C programmers. In *Proceedings of the 16th Australasian Computing Education Conference (ACE2014, Auckland, New Zealand, January, 20-23*, 51-57.

Fischer, G. (2000). Lifelong learning – More than teaching. *Journal of Interactive Learning Research, 11*(3/4), 265-294.

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representation of programs by novices and experts. *In CHI '93 Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (Amsterdam, The Netherlands)*, 74-79.

Ginat, D. (2000). Colorful examples for elaborating exploration of regularities in high-school CS1. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSEconference on Innovation and Technology in Computer Science Education (Helsinki, Finland)*, 81-84.

Guo, P. (2014, July 7). *Python is now the most popular introductory teaching language at top U.S. universities* [Blog post]. Retrieved August 2015 from http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext

Guzdial, M. (2008). Paving the way for computational thinking. *Communications of the ACM, 51*(8), 25-27.

Guzdial, M., & Weingarten, F.W. (1995). *Setting a computer science research agenda for educational technology, (CRA Report No. 1995)*. National Science Foundation.

Hayes, E. (2008). Girls, gaming, and trajectories of IT expertise. In Y.B. Kafai, C. Heeter, J. Denner, & J.Y. Sun (Eds.), *Beyond Barbie and mortal combat: New perspectives on gender and gaming* (pp. 217-230). Cambridge, MA: MIT Press.

Hundhausen, C. D., & Brown, J. L. (2007). What you see is what you code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing, 18*(1), 22-47.

Jansen, W. (2010). *Directions in security metrics research*. National Institute of Standards and Technology, Technical Report NISTIR 7564.

Jbara, A. & Feitelson, D. G. (2014). On the effect of code regularity on comprehension. In *Proceedings of the 22nd IEEE International Conference on Program Comprehension (June 2-3, Hyderabad, India)*, 189-200.

Kasto, N., & Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the Fifteenth Australasian Computing Education Conference (January 29 – February 1, Adelaide, Australia)*.

Kaufman, L., & Rousseeuw, P. J. (2009). *Finding groups in data: An introduction to cluster analysis*. Hoboken, NJ: John Wiley & Sons.

Kiesmuller, U. (2009). Diagnosing learners' problem-solving strategies using learning environments with algorithmic problems in secondary education. *ACM Transactions on Computing Education, 9*(3), article 17.

Kitchenham, B. (2010). What's up with software metrics? – A preliminary mapping study. *The Journal of Systems and Software, 83*(1), 37-51.

Kordaki, M. (2013). High school computing teachers' beliefs and practices: A case study. *Computers & Education, 68*, 141-152.

Lapidot, T. (2004). *The learning of computer-science teachers during their teaching work* (unpublished dissertation), Technion – Israeli Institute of Technology, Haifa, Israel.

Leach, R. J. (1995). Using metrics to evaluate student programs. *ACM SIGCSE Bulletin, 27*(2), 41-43.

Liberman, N., Ben-David Kolikant, Y., & Beeri, C. (2012). "Regressed experts" as a new state in teachers' professional development: Lessons from Computer Science teachers' adjustments to substantial changes in the curriculum. *Computer Science Education, 22*(3), 257-283.

Lukowiak, M., Radziszowski, S., Vallino, J., & Wood, C. (2014). Cybersecurity education: Bridging the gap between hardware and software domains. *ACM Transactions on Computing Education, 14*(1), article 2.

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: What's the connection? *Communication of ACM, 29*(7), 605-610.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality*. General Electric Company, Technical Report RADC-TR-77-369.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., & Euler, T. (2006). YALE: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Philadelphia, PA)*, 935-940.

Mihalcea, R., & Welch, C. (2015). What women want: Analyzing research publications to understand gender preferences in computer science. Presented at the *Workshop on Scholarly Big Data: AI Perspectives, Challenges, and Ideas as part of the 29th AAAI Conference on Artificial Intelligence (Austin, TX, January 25-30)*.

Moshtari, S., Sami, A., & Azimi, M. (2013). Using complexity metrics to improve software security. *Computer & Fraud Security, 2013*(5), 8-17.

Murphy, L., McCauley, R., Westbrook, S., Richards, B., Morrison, B. B., & Fossum, T. (2006). *Proceedings of SIGCSE '06 (Houston, TX, March 1-5)*, 17-21.

Ng, M.-H., Vee, C., Meyer, B., & Mannock, K. L. (2006). Understanding novice errors and error paths in Object-oriented programming through log analysis. In *Proceedings of Workshop on Educational Data Mining at the 8th International Conference on Intelligent Tutoring Systems (Jhongli, Taiwan)*, 13-20.

Pádua, W. (2010). Measuring complexity, effectiveness and efficiency in software course projects. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Cape Town, South Africa)*, 545-554.

Paechter, C. (2003). Power/knowledge, gender and curriculum change. *Journal of Educational Change, 4*(2), 129-148.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Peña-Ayala, A. (2014). Educational data mining: A survey and a data mining-based analysis of recent works. *Expert Systems with Applications, 41*(1), 1432-1462.

Pennington, N. (1987). Stimulus structures and mental representation in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295-341.

Perkins, D. (1999). The many faces of constructivism. *Educational Leadership, 57*(3), 6-11.

Phillips, D. C. (1995). The good, the bad, and the ugly: The many faces of constructivism. *Educational Researcher, 24*(7), 5-12.

Piaget, J. (2013). *The construction of reality in the child* (The international library of psychology). Oxon, UK: Routledge.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (February 29 – March 3, Raleigh, NC)*.

Prensky, M. (2007). How to teach with technology: Keeping both teachers and students comfortable in an era of exponential change. *Emerging Technologies for Learning, 2*, 40-46.

The Python Guide. (n.d.). *Section 9. Classes*. Retrieved 22 February 2014 from http://docs.python.org/2/tutorial/classes.html

Quinlan, J. R. (1996). Induction of decision trees. *Machine Learning, 1*(1), 81-106.

Richardson, V. (2003). Constructivist pedagogy. *Teachers College Record, 105*(9), 1623-1640.

Robles, G., & González-Barahona, J. M. (2013). Mining student repositories to gain learning analytics. In *Proceedings of the Fourth IEEE Global Engineering Education Conference (March 13-15, Berlin, Germany)*, 1249-1254.

Romero, C., & Ventura, S. (2010). Educational data mining: A review of the state of the art. *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews, 40*(6), 601-618.

Rowe, B. R., & Gallahar, M. P. (2006). Private sector cyber security investment strategies: An empirical analysis. *Fifth Workshop on the Economics of Information Security (Cambridge, UK)*, 1-23.

Sanner, M. F. (1999). Python: A programming language for software integration and development. *Journal of Molecular Graphics and Modelling, 17*(1), 57-61.

Savola, R. (2009). A security metrics taxonomization model for software-intensive systems. *Journal of Information Processing Systems, 5*(4), 197-205.

SEI CERT C++ Coding Standard. (2015). Software Engineering Institute, Carnegie Mellon University. Retrieved from https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637

Seals, C. D., Clanton, K., Agarwal, R., Doswell, F., & Thomas, C. M. (2008). Lifelong learning: Becoming computer savvy at a later age. *Educational Gerontology, 34*(12), 1055-1069.

Shin, Y., & Williams, L. (2013). Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering, 18*(1), 25-59.

Siemens, G., & Baker, R. (2015). Educational data mining and learning analytics. In R. K. Sawyer (Ed.), *Cambridge handbook of the learning sciences* (2nd ed.) (pp. 253-274). New York, NY: Cambridge University Press.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communication of ACM, 29*(9), 850-858.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, SE-10*(5), 595-609.

Stolfo, S., B ellovin, S. M., Keromytis, A. D., Sinclair, S., Smith, S. W., & Hershkop, S. (2008). *Insider attack and cyber security: Beyond the hacker*. Santa Clara, CA: Springer-Verlag.

Summerfield, M. (2010). *Programming in Python 3: A complete introduction to the Python language* (2nd ed.). Boston, MA: Pearson Education.

Taherkhani, A., & Malmi, L. (2013). Beacon- and schema-based method for recognizing algorithms from students' source code. *Journal of Educational Data Mining, 5*(2), 69-101.

Thompson, D., Bell, T., Adreae, P., & Robins, A. (2013). The role of teachers in implementing curriculum changes. *Proceedings of SIGSCE'13 (Denver, CO, March 6-9), 245-250.*

Truong, N., Roe, P., & Bancroft, P. (2005). Automated feedback for "fill in the gap" programming exercises. In *Proceedings of the 7th Australasian Computing Education Conference (Newcastle, NSW, Australia)*, 117–126.

Tryer, S. S. (2001). *A methodology for visually representing student C++ programming proficiency* (unpublished thesis). Texas Tech University, Lubbock, TX.

Vihavainen, A., Luukkainen, M., & Kurhila, J. (2013). Using students' programming behavior to predict success in introductory mathematics course. In *Proceedings of the 6th International Conference on Educational Dada Mining (Memphis, TN)*, 300-303.

Vilner, T., & Zur, E. (2006). Once she makes it, she is there: Gender differences in computer science study. *Proceedings of ITiCSE '06 (Bologna, Italy, June 26-28)*, 227-231.

Vujošević-Janičić, M., Nikolić, M., Tošić, D., & Kuncak, V. (2013). Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology, 55*(6), 1004-1016.

Wang, T., Su, X., Ma, P., Wang, Y., & Wang, K. (2011). Ability-training-oriented automated assessment in introductory programming course. *Computers & Education, 56*(1), 220-226.

Ward, J. H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association, 58*(301), 236-244.

Webb, D. C. & Miller, C. B. (2015). Gender analysis of a large scale survey of middle grades students' conceptions of computer science education. Presented at *Gender IT 2015 (Philadelphia, PA, April 24-25).*

Weiser, M., & Shertz, J. (1983). Programming problem representation in novice and expert programming. *International Journal of Man-Machine Studies, 19*(4), 391-398.

Whalley, J., & Kasto, N. (2014). How difficult are novice code writing tasks? A software metrics approach. In *Proceedings of the Sixtinth Australasian Computing Education Conference (January 20-23, Auckland, New Zealand)*.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies, 23*(4), 383-390.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.

Zaidman, M. (2004). Teaching defensive programming in Java. *Journal of Computer Sciences in Colleges, 19*(3), 33-43.

Biographies

Shlomi Boutnaru is a graduate student at the Department of Mathematics, Science and Technology Education, in the School of Education, Tel Aviv University (Israel). He holds a B.Sc. in Computer Science. He is involved in educational research in the fields of cyber security, programming languages and technology at large. Overall, he hopes to merge deep technological knowledge and novel pedagogical approaches. He is part of the team that wrote the cyber security curriculum for high schools in Israel.

Arnon Hershkovitz is a Senior Lecturer at the Department of Mathematics, Science and Technology Education, in the School of Education, Tel Aviv University (Israel). He holds a Ph.D. in Science Education, an M.A. in Applied Mathematics and a B.A. in Mathematics and Computer Science. His research interests lie at the intersection of education and technology. Many of his studies use methods from the fields of educational data mining and learning analytics. Among his research interests are: one-to-one computing in the classroom, learning/teaching processes in the social media era, student-teacher relationship, and genealogy as a unique lifelong learning experience in the information age.