Can the Security Mindset Make Students Better Testers? *

Sara Hooshangi
The George Washington University
Washington, DC 20052 U.S.A.
shoosh@gwu.edu

Richard Weiss
The Evergreen State College
Olympia, WA 98505 U.S.A.
weissr@evergreen.edu

Justin Cappos
New York University
Brooklyn, NY 11201, U.S.A.
jcappos@nyu.edu

ABSTRACT

Writing secure code requires a programmer to think both as a defender and an attacker. One can draw a parallel between this model of thinking and techniques used in test-driven development, where students learn by thinking about how to effectively test their code and anticipate possible bugs. In this study, we analyzed the quality of both attack and defense code that students wrote for an assignment given in an introductory security class of 75 (both graduate and senior undergraduate levels) at NYU. We made several observations regarding students' behaviors and the quality of both their defensive and offensive code. We saw that student defensive programs (i.e., assignments) are highly unique and that their attack programs (i.e., test cases) are also relatively unique. In addition, we examined how student behaviors in writing defense programs correlated with their attack program's effectiveness. We found evidence that students who learn to write good defensive programs can write effective attack programs, but the converse is not true. While further exploration of causality is needed, our results indicate that a greater pedagogical emphasis on defensive security may benefit students more than one that emphasizes offense.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; K.6.5 [**Security and Protection**]: Unauthorized access; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Security, Experimentation, Measurement

*This work was partially supported by NSF grants 1141341, 1223588, 1205415, 1241568 and 1241653.

SIGCSE'15, March 4-7, 2015, Kansas City, MO, USA.

ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677268

Keywords

Security; Python; Access Control; Testing

1. INTRODUCTION

Testing is an integral part of any real-world software development process. As such, software testing has become an integral part of computer science (CS) curricula in recent years [14, 4]. Researchers have extensively examined students' test-writing skills [16, 19, 7] and the primary focus has been on quantifying the quality of students' test codes, measuring their bug-revealing capabilities, and developing techniques to assess student-written software tests. Secure coding and software security testing, on the other hand, are relatively recent areas of exploration within the pedagogical community. In the wake of an increasing number of security breaches caused by software vulnerabilities, it has become essential that tomorrow's developers acquire crucial skills of writing secure code and developing software packages that can withstand malicious attacks.

In order to write secure code, students must learn how to think about system failure and software vulnerability, by developing what is often called a security mindset. They must learn how to think as an attacker and find ways to circumvent and exploit code flaws. In such a scenario, the program under attack is considered a *defense program* and the test case that is trying to break or exploit the defense program is an *attack program.* Writing programs that can defend against an attacker is at least as difficult as writing reliable programs. Acquiring the mindset to foresee potential vulnerabilities in a program is a valuable skill for students to learn.

We anticipate that including software security testing techniques as part of the typical software testing exercises used in CS classrooms will expand students' programming toolset and make them better equipped to tackle programming tasks. Software tests look for bugs that produce errors despite plausible input, whereas an attack program can trigger bugs with any possible input. A student who is able to write both defensive and attack programs is already thinking about ways to exploit vulnerabilities and ways to protect against possible breaches. Our study suggests that this more holistic approach may enhance students' programming skills and help them become more proficient in finding flaws in their own programs. The recent addition of IAS (Information Assurance and Security) to ACM/IEEE Computer Science Curricula 2013 [3] is a good starting point, but faculty need more examples of how to integrate these topics into their existing curriculum.

To test our hypothesis and to examine the overall effect of employing a security mindset approach in teaching CS, we examined students' abilities in writing both defensive and offensive programs, using a set of publicly available programming assignments [2]. Our study focused on both the quality of students' defense and attack programs, and the correlation between the two, in the context of a security assignment. Thus, we investigated more than traditional test-writing skills and quality of student-written tests. Specifically, students were first asked to write a defense program. After this, they were asked to write attack programs that would find a comprehensive set of bugs (including performance bugs and security related bugs) in the defense programs of all students in the class.

Using common techniques employed by the pedagogical community, such as the 'all-pair method ' [10, 9, 7], we explored students' ability to write resilient defensive programs, effective attack programs, and the relationship between the two skills. We observed the effectiveness of the security mindset approach on the overall performance of our students. Although some of our findings only reinforce what has already been shown by others in the context of students' test-writing skills, some of our results are unique to our experiment. Specifically, we observed the following outcomes using this particular attack-defense assignment, administrated to 75 students in a single class.

- In general, students wrote unique defense and attack programs.
- The number of attacks correlated with the quality of attacks. Thus, students who wrote more attack programs generally had more effective attacks.
- Some skills were indicative of students possessing other skills. For example, knowledge of how to scope variables appropriately correlated with the ability to write relatively bug free programs.
- Students who wrote good defense programs also wrote good attack programs. However writing good attack programs did not correlate with writing good defense programs.
- Students who could subsequently attack their own defense programs had written poor defensive programs.

In Section 2, we explain how our work relates to previous work on the pedagogy of testing. In Section 3, we describe the assignment, the class in which it was used, and how the data were collected. In Section 4, we present an analysis of our data and provide conclusions we draw from that data.

2. RELATED WORK

Over the past two decades, many CS educators have advocated for the integration of software testing into the CS curriculum [18, 11, 15]. Different strategies have been proposed for introducing software testing early on in undergraduate programs [13, 16, 10, 5, 17, 9]. Although introducing students to software testing is now common practice, assessing the quality of student-written tests has been a harder problem to address. The use of automated assessment tools [12, 5, 20, 6] and mutation analysis techniques [1, 19] have been examined as a way to assess the quality of student written tests. Most of these techniques measure 'code coverage ability' of student-written tests, but recent work has shown that such tools might produce an overestimation of student-written test quality [7].

Some have suggested that the quality of a test can be assessed by its ability to detect bugs or faults in a software program rather than by looking at its coverage of execution paths [7]. One technique for measuring the quality of testing is the 'all-pairs' technique proposed by Goldwasser in 2002 [10] and further developed by Edwards et al. [9, 7]. Using this approach, all student tests are run against all student programs.

How well do student written tests find significant bugs? Researchers have reported that students' tests are not high quality and that there is a large degree of similarity among student-written test programs. Moreover, most students are only able to find a small portion of bugs in a given program. Happy path testing–writing tests for typical scenarios–has been targeted as one reason that student-written tests tend to be quite similar [8].

3. PRELIMINARIES

3.1 The Assignment

Our observations in this paper are based on a two-part assignment [2] about access control *defense monitors.* A defense monitor, which runs as a user program, is similar to a reference monitor, which is an abstract machine that mediates all access to objects by subjects. Defense monitors can be used to allow, deny, or change the behavior of any set of calls. One critical aspect in creating a defense monitor is ensuring that it cannot be bypassed. This is one of the central concepts of computer security; successfully creating secure code requires a security mindset.

The assignment was given to students in two separate parts. In Part 1, students were asked to create a defense monitor to stop a user from reading data, writing over existing data, or writing new data to the end of a file if there was no permission to do so. This was more fine-grained than the usual Linux permissions. Students implemented five functions, as described in Table 1.

Function	Description
setread	Enable or disable reading from the file
setwrite	Enable or disable writing to the file
setappend	Enable or disable appending to the file
readat	Read data from the file if read permission is on
writeat	Overwrite existing data if write permission is on

Table 1: Summary of functions used in this assignment

In the assignment scenario, a user might have all, some, or none of the permissions enabled at any given time. By default, these permissions would be disabled. The readat function would allow the user to read data from the file if and only if the user has read permissions. The writeat function would allow the user to overwrite existing data if and only if write permissions are enabled. Similarly, the user could append new data to the file if and only if append permissions

were enabled. Attempts to overwrite or append data when the user does not have permissions to do so would result in nothing being written. The assignment provided students with an experience of thinking as a defender and reinforced concepts of how to build a secure system.

Figure 1 shows a code snippet from the [inadequate] defense monitor that was provided to the students.

```
def setread(self,enabled):
    mycontext['read'] = enabled

def readat(self,bytes,offset):
    if not mycontext['read']:
      raise ValueError
    return self.file.readat(bytes,offset)
```

Figure 1: An excerpt from the example (inadequate) defensive program students were given.

Students were asked to adhere to three design criteria when writing their defense monitor programs:

- Precision: The security layer should only stop certain actions from being blocked. All other actions should be allowed even when combined with blocked actions. For example, if a user tries to overwrite data and append new data in the same write, and has permissions to write, but not append, then the write should succeed and the append should fail. Thus, data would be overwritten up to the end of the file, but nothing would be appended.

- Efficiency: The security layer should use a minimum number of resources so that performance is not compromised. The code should also avoid actions such as re-reading a file before each write.

- Security: The attacker should not be able to circumvent the security layer. Hence, if any data can be written without the corresponding permission, security will be compromised.

In Part 2, each student wrote one or more attack programs to circumvent the defense monitors written by others in Part 1. The second part of the assignment was not given until after all students had submitted their defense monitors, which were then made available to the entire class. This allowed students to view security from an attacker's standpoint and to think about how to trigger failures. A complete module write-up for each part, with detailed instructions (and instructor-only solutions), is available publicly online from the assignment creators [2].

3.2 The Population

This assignment has been used in a dozen classes at four different institutions. For this study, we selected the class with the largest number of students. In Fall 2013, 75 students completed the assignment described above for an 'Introduction to Security' course at NYU. This course is a senior/first-year graduate level class designed for Computer Science BS / MS students and also for Cybersecurity MS students. All programs were written in a subset of Python [2]. The students were given one week to complete each part. In order to make the assignments easier to complete, especially for students with no experience with Python, we gave the students a template for a simple defense monitor that inadequately checks for a desired security property. Students submitted their defense monitors to complete Part 1 of the assignment. Each student submitted one monitor and we received a total of 75 monitors. To ensure that student monitors were free of syntax errors, all monitors were run against an instructor-written attack test to check for validity. A total of 61 working, error-free monitors were identified. Students were strongly warned about consequences of cheating and all submitted programs were checked for evidence of cheating by examining code similarity as well as behavioral similarity. We removed student assignments that showed evidence of cheating, but we cannot be certain we caught all instances.

After students completed the defensive assignment (Part 1), they were given a copy of all the students' defensive programs and told to write attack programs that would bypass as many defense programs as possible (constituting Part 2 of the assignment). Students were encouraged to write each attack case as a separate file because any test that produced incorrect results for an instructor-written defense monitor would be discarded entirely. A total of 444 attack test files were submitted. The attack programs were first run against the instructor-written defense monitor to remove incorrect attacks. All of the attack programs that printed that they bypassed the instructor's defense monitor were manually inspected and none were found to be correct. 68 students produced viable attacks and a total of 325 viable attack files were found. The number of attack files submitted by each student varied from 1 to 14, with a median of 5. After careful examination of the attack files, it was determined that 28 files were identical to the sample attack file that the instructor had given the students as a guide.

4. RESULTS AND ANALYSIS

To analyze students' ability to attack and defend, we used an 'all-pairs testing' strategy that has been shown to be effective [10, 19, 9]. Our automated script ran each attack test file against each defense monitor, for a total of 19,825 trials. An attack test was flagged as being successful for a specific instance if the attack test was able to cause an error in the defense monitor or to bypass its defense. The outcome of each instance was recorded in a 2D matrix, which helped us keep track of all attack/defense pairs. Below we describe our findings and data analysis.

4.1 Were Student Submissions Unique?

The first area we explored was whether or not student submissions were primarily unique or similar to others. For example, did many students write defensive programs that had the same flaws? Did submitted attack programs effectively test the same flaws?

Methodology. To examine the variations in students programs, we first looked for uniqueness in individual attack tests. Two tests were considered different if there was at least one defense monitor for which they gave different results.

We observed that there were 150 unique tests; they had an attack that differed from every other attack on at least one monitor. We also looked at individual students' work and observed that 47 students (out of 68) had produced a

testing pattern that was different from the rest of the class. These results show that variations exist both in the individual programs and the collected test functionality provided by a student's attack programs. We also analyzed similarities across the defense monitors. Two monitors were considered different if they behaved differently in response to at least one student attack test. In the 61 working monitors submitted, 49 produced unique responses to the student attacks.

Conclusion. *We find that student submissions for defenses have a high degree of uniqueness*, with about 80% being unique. This shows that secure coding is a complex problem and different students find different solutions.

However, we found that just over half (51%) of the attack programs were unique. We think the reason for a lower number of unique test cases was due to two factors. First, students were encouraged to submit individual, separate tests for small cases. Thus, many students may have decided to just submit the same basic test provided by the instructor. (In fact, dozens of students submitted one of the trivial example programs we had provided on the assignment sheet.) Second, there were many more attack programs than defense monitors, with most students (90%) submitting multiple attack programs. Since there were 325 attack programs, there was a higher chance of non-uniqueness. We note that 78% of all students had at least one unique test in their submissions. These factors are evidence that *attack programs submitted by students are also very unique.*

4.2 Do Multiple Attacks Benefit Performance?

Given that most students wrote several unique attack programs, it is natural to ask what we can learn from their submissions. We investigated to see if there was a common pattern that would suggest which students benefitted from multiple attack submissions and which ones did not. We also explored the more specific question of whether the total number of submissions was correlated with attack effectiveness.

Methodology. To examine the possible benefit of writing multiple attack cases, we first measured the quality of student attacks by defining a Student Attack Quality (SAQ score) as a student's overall ability to attack all monitors. For students who submitted more than one attack file, a student's effort was considered a successful attack against a monitor if at least one of her/his attack files was able to break that particular monitor. The overall SAQ score for the student was then calculated as the sum of all successful attacks divided by the total number of monitors. Using this data, we were able to look at whether students who wrote more tests did better. Figure 2 summarizes, in one graph, the results of our analysis. In this graph, students are sorted according to the number of tests that they wrote (secondary y-axis), and within each category students are sorted according to their overall SAQ score (primary y-axis). We can see that there was a wide range in SAQ scores–from 23% to 92%. This shows a wide range in testing skills and strategies. Also, students who wrote between four and six tests had the highest overall SAQ scores. We found that there was a weak correlation between the number of tests and SAQ score (correlation of r = 0.63, data not shown).

We next looked at how individual tests contributed to a student's overall SAQ score. We wanted to know whether students partitioned the attack space into equal parts and wrote an attack for each part or whether they had one attack

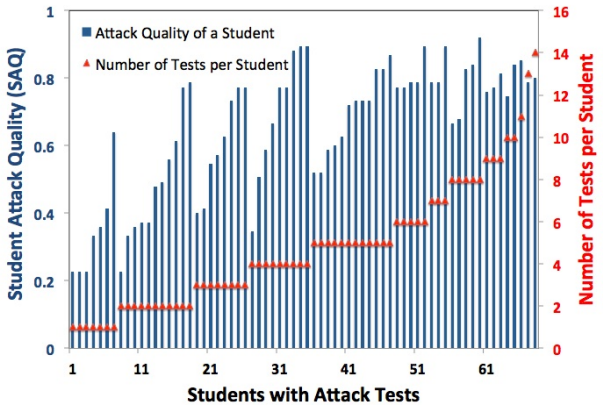

Figure 2: Students' quality of attack versus number of tests submitted. X-axis represents students who submitted at least one attack program. Left y-axis represents the SAQ score for each student and the right y-axis represents the number of tests submitted by each student.

script that accounted for most of their total, but added additional scripts that played only a minor role. This analysis is captured in Figure 3, which compares a student's overall SAQ score with the maximum score for a single script for that particular student.

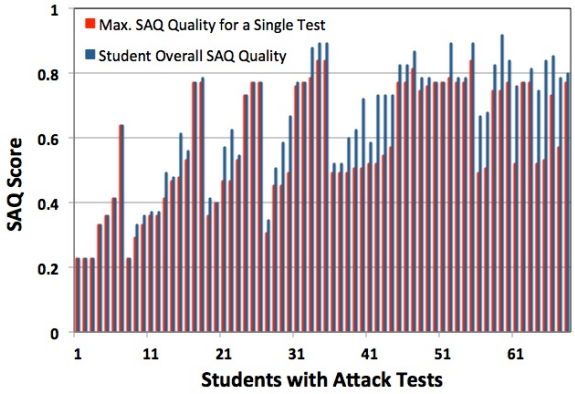

Figure 3: The benefit of having multiple attack tests. The overall SAQ score (blue bar) and max SAQ scores on a single test (red bar) are plotted for all 68 students. Students are sorted on the x-axis by the number of tests per student.

Conclusion. We can see from Figure 3 that the maximum score is usually very close to the overall score, suggesting that students wrote one script that accounted for most of their points. At first glance, this seems to be at odds with the first result that students who wrote more tests did better. However, there are two different types of attacks that students wrote: those that focused on specific types of errors, and those that focused on a particular feature or capability addressed by the monitor. The former turned out to be much more powerful for finding vulnerabilities. For example, if the monitor made the implicit assumption that

only one file would be open at a time, it was very likely to have a bug that would be triggered by opening multiple files. Writing an attack script to trigger this bug worked for a large number of monitors. It seems that students who had this realization were also thorough and checked several features of the monitor, e.g., whether it handled read permissions correctly, whether it handled write permissions correctly, etc.

4.3 What Accounts For The Difference Between Max and Overall SAQ?

We explored the source of the differential between maximum (max) and overall SAQ scores in student attacks. We first studied whether the diversity of attack programs might contribute to students' low differential between max and overall SAQ scores before zeroing in on a primary reason.

Methodology. The class instructor looked through the attacks written by the 10 students with the highest difference between their max and overall SAQ scores (and thus benefited the most from additional tests) and the 10 students who had the lowest difference between their max and overall SAQ (who received no benefit from additional tests). Without knowing which were which, the instructor tried to rate the level of diversity in the student attack programs. For example, submitted attack programs that used different calls/permissions patterns were considered diverse because they would be likely to touch different code paths in defense monitors. (The detailed results of this categorization are omitted for space reasons.) 18 of the submissions were rated to have a reasonable degree of diversity (all 10 with the highest differential were in this category.)

Conclusion. Students who did well wrote diverse tests (as one may expect). However, many students who exhibited no benefit from any additional tests also wrote diverse tests. We concluded that *diversity was not sufficient to determine whether additional attacks were useful.*

With further investigation, we identified the main contributing factor to be when a student wrote a multi-file attack program and did not generate other attacks of high quality. In our data set, some students wrote a defense monitor that incorrectly placed information about permissions in the global scope instead of treating them as a per-file property. In our pool of student defenses, those defenses that did correctly scope the permissions data to a file instance also did not contain simple mistakes in the monitor. Thus *the understanding of how to appropriately scope variables was an indicator of student ability to write reasonably error free code for the remainder of the assignment.*

4.4 Are Attack/Defense Abilities Correlated?

Another question to emerge from our work is to what extent attack and defense abilities are correlated. For example, does skill in one indicate skill in the other?

Methodology.

We compared attack ability with student defense ability. We first defined Defense against Attack Quality (DAQ score) as the measure of a defense monitor's resistance against students' attack programs. The DAQ score for a defense monitor was calculated as the ratio of the number of students who were not successful in attacking a particular monitor divided by the total number of students who had viable attack programs (68 in our case). The SAQ scores were then plotted against DAQ scores, as shown in Figure 4. We labeled the student points on this graph based upon whether the student submitted an attack that could bypass their own defense monitor.

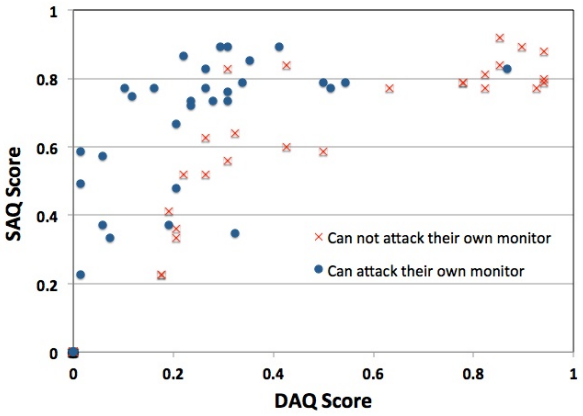

Figure 4: Quality of attack as a function of quality of defense. Blue circles indicate students who are able to break their own monitors and red x's indicate students who are not able to bypass their own monitors.

Conclusion. Our observations here bring us back to the security mindset to reveal something that was not clear at first. The security mindset requires the ability to trigger failure modes, but when it comes to secure coding, it also requires an ability to recognize and reflect on assumptions. There were a number of students who were able to attack other reference monitors, and they seem to have a part of the security mindset, but only some of them were able to draw on that mindset to write secure programs. Moreover, we saw that students who were able to attack their own defense monitors were the least skilled at applying the security mindset in general and that their attacks were also the weakest. A stronger measure of the security mindset is whether one can write secure code.

Our findings show that *students' defensive ability is indicative of their ability to attack.* Having the ability to defend entails being able to consider, and handle, possible attacks. However, *attack ability is not indicative of talent with defense.* Thus, a student may be a capable attacker, yet not be able to write defensive programs. We plan to further investigate this point in future work.

We also found that *students who were able to break their own defensive programs typically wrote very poor defensive programs.* One might be tempted to conclude that a student's ability to self-attack may serve as a meter of their abilities. However, since students are given each other's defensive programs to examine when writing attacks, their observations of other students' defensive programs may bias the findings.

5. CONCLUSIONS AND FUTURE WORK

We have examined the effect of teaching the security mindset by analyzing students' writing of attack and defense monitors. In our study, both students' defense and attack programs showed a high degree of uniqueness. This stands in contrast to observations that students tend to write test

codes similar to their classmates when they are testing their own code. However, diversity alone does not guarantee attack test quality. In addition, we found that students who wrote good defense monitors also wrote good attack tests. These results suggest that the defense monitor exercise that we used, and similar strategies, could be useful in improving the quality of tests that students write. However, our experiments do not prove causation. In the future we would like to analyze whether there is a causal relationship. Another question that we would like to test using our quantitative framework is whether students who have written one defense monitor do better when writing another one that has significantly different content.

The assignment we used is not the only way to teach the security mindset, and that opens up the possibility that there may be other exercises that would improve testing and secure coding skills. In future work we intend to explore similar assignments in lower level classes. We also hope to perform a follow up study where we examine how student attack and defense abilities carry over in classes that specialize in these topics.

6. REFERENCES

[1] K. Aaltonen, P. Ihantola, and O. Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '10, pages 153–160, New York, NY, USA, 2010. ACM.

[2] J. Cappos and R. Weiss. Teaching the security mindset with reference monitors. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 523–528, New York, NY, USA, 2014. ACM.

[3] Computer science curricula 2013. `http://www.acm.org/education/CS2013-final-report.pdf/`.

[4] C. Desai, D. S. Janzen, and J. Clements. Implications of integrating test-driven development into CS1/CS2 curricula. *SIGCSE Bull.*, 41(1):148–152, Mar. 2009.

[5] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 26–30, New York, NY, USA, 2004. ACM.

[6] S. H. Edwards and M. A. Pérez-Quiñones. Experiences using test-driven development with an automated grader. *J. Comput. Sci. Coll.*, 22(3):44–50, Jan. 2007.

[7] S. H. Edwards and Z. Shams. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 354–363, New York, NY, USA, 2014. ACM.

[8] S. H. Edwards and Z. Shams. Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 171–176, New York, NY, USA, 2014. ACM.

[9] S. H. Edwards, Z. Shams, M. Cogswell, and R. C. Senkbeil. Running students' software tests against each others' code: New life for an old 'gimmick'. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 221–226, New York, NY, USA, 2012. ACM.

[10] M. H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. *SIGCSE Bull.*, 34(1):271–275, Feb. 2002.

[11] T. B. Hilburn and M. Townhidnejad. Software quality: A curriculum postscript? *SIGCSE Bull.*, 32(1):167–171, Mar. 2000.

[12] D. Jackson and M. Usher. Grading student programs using assyst. *SIGCSE Bull.*, 29(1):335–339, Mar. 1997.

[13] U. Jackson, B. Z. Manaris, and R. A. McCauley. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 360–364, New York, NY, USA, 1997. ACM.

[14] D. S. Janzen and H. Saiedian. Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum. *SIGCSE Bull.*, 38(1):254–258, Mar. 2006.

[15] E. L. Jones. Software testing in the computer science curriculum – a holistic approach. In *Proceedings of the Australasian Conference on Computing Education*, ACSE '00, pages 153–157, New York, NY, USA, 2000. ACM.

[16] E. L. Jones. Integrating testing into the curriculum-arsenic in small doses. *SIGCSE Bull.*, 33(1):337–341, Feb. 2001.

[17] W. Marrero and A. Settle. Testing first: Emphasizing testing in early programming courses. *SIGCSE Bull.*, 37(3):4–8, June 2005.

[18] R. McCauley and U. Jackson. Teaching software engineering early - experiences and results. In *Frontiers in Education Conference, 1998. FIE '98. 28th Annual*, volume 2, pages 800–804, Nov 1998.

[19] Z. Shams and S. H. Edwards. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 53–58, New York, NY, USA, 2013. ACM.

[20] J. Spacco and W. Pugh. Helping students appreciate test-driven development (tdd). In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 907–913, New York, NY, USA, 2006. ACM.