Does Creating Programming Assignments with Tests Lead to Improved Performance in Writing Unit Tests?

Vilma Kangas
University of Helsinki
Helsinki, Finland
vilma.l.kangas@helsinki.fi

Nea Pirttinen
University of Helsinki
Helsinki, Finland
nea.pirttinen@helsinki.fi

Henrik Nygren
University of Helsinki
Helsinki, Finland
henrik.nygren@helsinki.fi

Juho Leinonen
University of Helsinki
Helsinki, Finland
juho.leinonen@helsinki.fi

Arto Hellas
University of Helsinki
Helsinki, Finland
arto.hellas@helsinki.fi

ABSTRACT

We have constructed a tool, CrowdSorcerer, in which students create programming assignments, their model solutions and associated test cases using a simple input-output format. We have used the tool as a part of an introductory programming course with normal course activities such as programming assignments and a final exam.

In our work, we focus on whether creating programming assignments and associated tests correlate with students' performance in a testing-related exam question. We study this through an analysis of the quality of student-written tests within the tool, measured using the number of test cases, line coverage and mutation coverage, and students' performance in testing related exam question, measured using exam points. Finally, we study whether previous programming experience correlates with how students act within the tool and within the testing related exam question.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Information systems** → **Crowdsourcing**; • **Human-centered computing** → *Collaborative content creation*; • **Social and professional topics** → *Computing education*;

KEYWORDS

testing, crowdsourcing, assignment creation, educational data mining

ACM Reference Format:
Vilma Kangas, Nea Pirttinen, Henrik Nygren, Juho Leinonen, and Arto Hellas. 2019. Does Creating Programming Assignments with Tests Lead to Improved Performance in Writing Unit Tests?. In *ACM Global Computing Education Conference 2019 (CompEd '19), May 17–19, 2019, Chengdu, Sichuan, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3300115.3309516

CompEd '19, May 17–19, 2019, Chengdu, Sichuan, China

ACM ISBN 978-1-4503-6259-7/19/05…$15.00
https://doi.org/10.1145/3300115.3309516

1 INTRODUCTION

Testing "is not a glamorous topic within software engineering" [7]. There seems to be a consensus on this in the field – testing is traditionally treated as the boring and unpleasant part of a software project, and there are often not enough time nor resources provided to do it well, let alone comprehensively [3, 18]. There is no general agreement in software engineering education and computer science education on to what extent one should integrate testing into introductory programming courses, even though the topic is considered important.

For example, Lappalainen et al. [14] suggest writing input-output test cases into method comments, Edwards [10] suggests having students write unit tests and providing students' feedback on the tests that they wrote, including test coverage, and Pirttinen et al. [19] suggest having students create problems with input-output test cases. At the same time, successful integration of testing practices into introductory programming courses is not trivial. For example, when integrating unit testing into first-semester programming classes, Barriocanal et al. found that only approximately 10% of the students actually wrote unit tests [1]. Interest and motivation also plays a role – Carrington suggests that students find writing tests for programs developed by other people more constructive than finding faults in their own programs [4].

In this study, we look at an approach where software testing – without mentioning software testing – is introduced to students in a first-semester introductory programming course through the use of CrowdSorcerer, a tool in which students come up with programming assignments and their associated tests. Using the tool, first described by Pirttinen et al. [19], we ask the students to create programming assignments to other students in the class and to generate simple input-output test cases for the assignments that they are creating. While the course does not initially discuss testing at all, the final course week provides a worked example on unit testing, and refers students back to the programming assignments and test cases that the students themselves have written.

We are interested if the use of the simple assignment generation tool in which students both write the assignment and the tests helps students learn testing. That is, whether experience gained from assignment and test generation transfers to other testing related activities. To evaluate this, we study students' performance in a computer-based exam where students are expected to implement

unit tests for a given class, measured using manually graded exam points. We study how automatically extractable metrics such as test count, line coverage and mutation coverage correlate with students' performance in the testing related exam question, and also study whether students' previous programming background contributes to their behavior in the tool and the exam.

This article is organized as follows. First, in Section 2, we discuss related work. Then, in Section 3, we introduce the research questions and describe the methodology of the study. We present the results in Section 4 and discuss their implications and the limitations of the study in Section 5. Finally, conclusions and future work are presented in Section 6.

2 BACKGROUND

We first discuss research on ways of teaching testing and how to assess the quality of students' tests. Then, we briefly discuss how testing could be incorporated into introductory programming courses. Finally, we discuss research on influence of previous programming experience on performance in programming courses.

Testing is an important concept to teach, even if students do not like the increased workload and time spent on creating test cases. One way to make the learning process more fun and engaging is using tools or games for teaching testing. Mutation testing game Code Defenders by Clegg et al. [6] teaches software testing concepts such as statement or branch coverage through gameplay. CodeWrite [9] teaches students testing by first letting them write their own programming exercises and then a set of test cases for those exercises. Similarly, CrowdSorcerer [19] can be used to crowdsource programming assignments while teaching testing practices and encouraging students to read each others' code through a peer review functionality. Studies have shown that these kinds of exercise generation activities in programming courses can improve exam performance and help learning programming [8, 15, 16].

When investigating ways to teach testing, a question of how to evaluate the quality of student-written tests arises. CodeWrite and CrowdSorcerer both rely on peer-assessment [9, 19], but automated measures also exist. Edwards et al. [11] conducted an experiment with three test quality measures for assessing programs written by students in a data structures course: composite code coverage (counts how many of the methods, statements and branches of the code are executed), all-pairs scores (students' tests are run against the programs of others) and mutant kill ratios (creates mutants, where artificial defects are inserted into the program, and then runs student's test suite to see if it can distinguish the mutated version from the original program). The all-pairs score correlated the best with how well students' tests could reveal a bug.

A study by Marrero and Settle [17] investigated benefits of placing greater emphasis on testing in programming assignments in introductory programming courses. Their study did not find any uniform improvement in student performance, and concluded that especially novice programmers can struggle with the idea of handling assignments in several parts (program itself and its tests). Students in the group with more emphasis on testing seemed to struggle less with the abstraction that involved designing and implementing classes. Emphasis on testing also forced the students to concentrate on good software engineering practices even beyond testing, such as interaction with colleagues and peers, and appreciation of clear requirements. Jansen and Saiedian [13] concluded their study with similarly inconclusive results, stating that while test-driven learning in introductory programming courses does have its benefits in adopting good testing practices early on, especially student responses highly favoured test-last approach because of apparent increased workload of test-first approach.

Teachers often integrate at least some testing practices into introductory programming, though the level of emphasis varies greatly. At best, students can be very accepting to including for example unit testing practices to programming assignments, and think that the effort spent on them is worthwhile and has a positive impact on the quality of their code [1]. A study by Edwards [10] suggests that teaching test-driven development in introductory programming courses can improve both the programming and testing skills of the students. Whether one should start with test-driven development or gradually increase testing remains an open question.

Prior programming experience seems to have a positive impact on students' learning and performance in programming courses [12]. The more programming languages a student has learned before taking an introductory programming course, the greater the effect. These observations on the impact of prior experience are supported by several researchers, for example Wiedenbeck et al. [22] noted that factors related to programming experience had a weak but significant correlation with introductory programming course scores. The experience does not necessarily need to be even related to programming, but to computer use in general. For example, Wilson and Shrock [23] note that even factors such as the use of internet, time spent on gaming, and the use of productivity software explain some of the variance in the score of an introductory programming exam [23]; this suggests that mere experience from the use of a computer can be beneficial for learning programming.

However, some studies have noted contradicting results. For example, Watson et al. [21] found that while previous programming experience had significant positive impact on introductory programming course scores when compared to those with no experience at all, there was a weak but statistically insignificant negative correlation with course points and programming years – that is, prior experience may contribute negatively to students' effort. Similarly, Bergin and Reilly [2] found no statistically significant difference between students with or without previous programming experience, and that students with no previous experience had marginally higher mean overall score on an introductory programming course.

3 METHODOLOGY

3.1 Context and tool description

Our study was conducted in spring 2018 in an introductory programming course organized in Java at University of Helsinki. The course offered help in walk-in laboratories with the option of doing the course fully online. The material consists of 7 weeks of content and programming exercises that go through the basics of Java programming, starting from basic procedural programming and ending with building programs that use several classes with different responsibilities ranging from a textual user interface to methods for storing data.

The course also lets the students write their own programming assignments using CrowdSorcerer [19]. Within CrowdSorcerer, students are instructed to create programming assignments in the following manner: First, create an assignment handout that is related to the given topic, for example if-else-statements. Then, write a model solution for the assignment and adjust it to create a code template that can be given to others. Finally, write a set of test cases for the assignment. The test cases are written as pairs of inputs and outputs: for a given input, the output of the program is expected to be the given output or to contain parts of the given output. The students were required to write at least one test case for each programming assignment they created. In our context, creating assignments with the tool was voluntary. The assignments created by students were peer reviewed by other students on the course in subsequent weeks of the course (see [20] for more details); students had three opportunities to create new assignments, during week 2, week 4 and week 6 of the 7 week course.

Creating assignments with CrowdSorcerer has many potential benefits in addition to having the students think about how their program needs to be tested. When students peer review other students' assignments, they get to look at source code written by others. This can have benefits such as students possibly understanding why clean code is important for code maintenance, as well as learning alternative ways to approach a programming problem. Additionally, some of the assignments created by students can be integrated into subsequent course iterations.

The last week of the course material discussed testing and provided a worked example of creating a set of unit tests, which the students were expected to follow. The material referred back to the assignment generation problems with CrowdSorcerer.

3.2 Research questions

We study whether the tool has benefit to students' skills in testing their programs. The research questions for this study are as follows:

RQ1. Are students who use the tool more likely to answer testing-related exam questions?

RQ2. Does the quality of the tests the students create with the tool predict the quality of their answer to testing-related exam questions?

RQ3. How does students' previous programming experience influence their willingness to use the tool?

RQ4. Does students' previous programming experience contribute to the quality of written tests?

The first research question is answered through a statistical analysis of the usage of the tool and students' performance in a testing-related question in our course exam. The second question is answered through an analysis of student-generated test quantity, line coverage, and mutation coverage, and their correlation with the test-related exam question. The third and fourth research questions look into whether students' previous programming background contribute to students' tool usage or the quality of the written tests.

3.3 Data

CrowdSorcerer collects student identifiers, created programming assignments and tests, and time stamps of events within the tool. Besides this data, we use exam results, focusing on a question where the students were expected to implement unit tests for a given program. The exam question is provided in Appendix A.

# Assignments	# Students	% Answered	Avg. points (sd)
None	168	97.6%	2.78 (1.54)
1-2	80	97.5%	2.92 (1.54)
3	22	100%	3.47 (1.08)

Table 1: Testing exam question performance categorized based on the number of generated programming assignments. # Assignments column indicates the number of generated assignments, # Students column the number of students in that category, % Answered the percentage of students in that category who answered to the exam question on testing, and Avg. points (sd) represents the average points and standard deviation from the testing-related exam question (on a scale from 0 to 4).

The data consists of students who participated in the exam and gave a permission to use their data for research purposes ($n = 270$). When calculating particular values, for example the average points received from the testing-related exam question, only students who answered the question were included. If a student retook the exam, for example to raise their course grade, only the first exam attempt was investigated. Finally, when calculating correlations between performance of the tests written in the tool and the points received from the testing-related exam question, only those who had created at least one assignment and answered the testing-related exam question were studied.

4 RESULTS

4.1 Usage and exam question performance

In total, 270 students gave permission for the collection and usage of their data. From these students, 22 used CrowdSorcerer to create assignments every time it was visible in the material, 80 used it 1 to 2 times, and 168 did not use it at any point of the course. When calculating usage, we count only the times when a student created a new programming assignment for a different problem – repeated returns for the same problem are excluded from analysis. From the 270 students, 264 answered the testing-related exam question, 100 of whom had used the tool to create an assignment and 164 had not. 97.5% of the students who used the tool 1 to 2 times attempted the testing exercise in the exam, whereas 97.6% of those who did not use the tool answered to the testing exercise.

On average, those who did not use CrowdSorcerer received 2.78 points from the testing-related exercise. Those who used the tool once or twice received an average of 2.92 points, and the average was 3.47 points for those who used it every time. The standard deviations were 1.54, 1.54 and 1.08 for these groups respectively. These results can also be seen in Table 1.

We studied if the number of assignments created using CrowdSorcerer correlated with the points that the student received from the testing-related exam question. Using Kolmogorov–Smirnov test, we compared the exam results of the groups: no assignments created; 1–2 assignments created; and 3 assignments created. The test, when corrected for multiple comparisons using Bonferroni correction, showed no statistically significant difference between the groups.

4.2 Time and exam performance

Time spent using CrowdSorcerer was estimated based on the time stamps collected by the tool. The students were divided into three groups of approximately equal sizes based on the time spent with the tool. The first group contains those who used the tool for less than half a minute, since most students did not use the tool at all. The rest were divided into two groups based on the median of the usage times, which was around 17 minutes, so that the first group contains the students who spent 0.5 to 17 minutes using the tool, and the students in the last group spent more than 17 minutes.

In the first group, 95.1% had answered the testing-related exam question. In the second group, the response rate was 100%, and in the last group, it was 99%. The average points received from the question were 2.68 for the first group, 2.81 for the second group, and 3.12 for the last group, with the standard deviations 1.52, 1.58 and 1.44, respectively. These results can be seen in Table 2.

Time	# Students	% Answered	Avg. points (sd)
< 0.5 minutes	102	95.10%	2.68 (1.52)
0.5 - 17 minutes	68	100%	2.81 (1.58)
> 17 minutes	100	99%	3.12 (1.44)

Table 2: Testing exam question performance categorized based on the amount of time spent in the tool. Time column indicates the time spent on generating the assignments, # Students column the number of students in that category, % Answered the percentage of students in that category who answered to the exam question on testing, and Avg. points (sd) represents the average points and standard deviation from the testing-related exercise (on a scale from 0 to 4).

We studied whether the time spent on CrowdSorcerer correlated with the points that the student received from the testing-related exam question. Using Kolmogorov–Smirnov test, we compared the exam results of the groups: less than 0.5 minutes in the tool; 0.5–17 minutes in the tool; and over 17 minutes in the tool. Kolmogorov–Smirnov test, when corrected for multiple comparisons using Bonferroni correction, showed no statistically significant difference between the groups – the test comparing the less than 0.5 minutes in the tool and the over 17 minutes in the tool groups had $p = 0.017$.

4.3 Testing effort and exam performance

Next, we calculated the correlations between the points received from the testing-related question in the exam and the test performance when using the tool. Three metrics were used to evaluate the testing effort: (1) number of test cases, (2) line coverage, and (3) mutation coverage.

We used Spearman's rank correlations, as the relationship between test performance and exam question points is not necessarily linear. For example, it could be that the increase from 0% coverage to 50% is more meaningful than the increase from 50% coverage to 100% coverage. The correlations and p-values can be seen in Table 3. All of the correlations are weak and not statistically significant.

4.4 Influence of programming experience

At the beginning of the course, students were asked about their previous programming experience. Out of the 270 students who had given consent for the study, 141 provided details on their programming background. From these 141 students, 28 had no previous experience at all, 50 had programmed for 6 to 49 hours, and 63 for over 50 hours. We organized these students into four groups: "did not report programming experience", "no programming experience", "some programming experience" and "a lot of programming experience", respectively.

	Course exam points
# Test cases	$r = 0.093$, $p = 0.372$
Line coverage (pct)	$r = -0.086$, $p = 0.400$
Mutation coverage (pct)	$r = 0.143$, $p = 0.162$
Time spent on the tool	$r = 0.184$, $p = 0.072$

Table 3: Spearman correlations and p-values between the testing effort – measured using the number of test cases, line coverage, mutation coverage and time spent – and points received from the testing-related question in the exam.

From those who did not report programming experience, 40.6% completed at least one assignment with CrowdSorcerer. The rates were 39.3% (novices), 36.0% (some programming experience) and 33.3% (a lot of programming experience). The average time spent on the tool was 23.2 minutes with a standard deviation of 42.8 (did not report programming experience), 20.8 minutes with the standard deviation of 37.2 (novices), 38.5 minutes with the standard deviation of 114.1 (some experience), and 25.9 minutes with the standard deviation of 50.1 (a lot of programming experience).

The average amounts of completed assignments were: 1.60 (no reported programming experience), 1.27 (no experience), 1.39 (some experience), and 1.86 (a lot of experience). Standard deviations were 1.79, 1.41, 1.55 and 2.08, respectively.

100% of those with at least some programming experience had answered the testing-related question in the course exam, whereas 96.4% of those with no experience and 96.9% of those who did not report programming experience answered the question. Averages of points received from the question were 2.55 (did not report programming experience), 2.93 (no experience), 2.86 (some experience), and 3.05 (a lot of experience), and the standard deviations were 3.12, 3.37, 3.28 and 3.45, respectively. The difference between these populations is shown in Table 4.

We studied whether those with more programming experience were more likely to use the tool. Using Kolmogorov–Smirnov test, we compared the time spent on the tool as well as the number of created assignments with it for the three groups: those who had no programming experience, those with some experience, and those with a lot of experience. The test showed no statistically significant difference between the groups.

5 DISCUSSION

5.1 Tool usage and student performance

When analyzing the usage of the tool, most of the participants (62%) chose not to use the tool at all, while only 8% of the population used it every time it was in the material. This reflects previous results in teaching testing to novices, for example Borriocanal et al. [1] observed that only 10% of their students wrote unit tests.

	Did not report programming experience	No programming experience	Some programming experience	A lot of programming experience
Participated in the course exam	129	28	50	63
Created at least one assignment with the tool	40.6%	39.3%	36.0%	33.3%
Average and SD of time spent on the tool (in minutes)	Average: 23.2 SD: 42.8	Average: 20.8 SD: 37.2	Average: 38.5 SD: 114.1	Average: 25.9 SD: 50.1
Average and SD of the number of completed assignments with the tool	Average: 1.60 SD: 1.79	Average: 1.27 SD: 1.41	Average: 1.39 SD: 1.55	Average: 1.86 SD: 2.08
Answered the testing-related exam question	96.9%	96.4%	100%	100%
Average and SD of the points from the testing-related exam question	Average: 2.55 SD: 3.12	Average: 2.93 SD: 3.37	Average: 2.86 SD: 3.28	Average: 3.05 SD: 3.45

Table 4: Influence of previous programming experience for those who participated in the course exam. Points from the testing-related question are scaled from 0 to 4. In the columns, "no programming experience" means that the student reported less than five hours of previous programming experience, "some" means 6-49 hours of experience, and "a lot" more than 50 hours. Usage of the tool is counted on separate occurrences, not from repeated attempts of the same task. The average time is calculated based on all students in the group, not only those who used the tool. SD means standard deviation.

Our results do not indicate that students who use the tool more would be more likely to correctly answer the testing-related question in the course exam. However, it still seems that the more time the students spend using CrowdSorcerer and the more assignments they created, the better they performed in the testing-related exam question. Not creating assignments led to an average of 2.78 points from the exam question, while always using the tool to create an assignment led to an average of 3.47 points. Similarly, those who used the tool for less than half a minute got an average of 2.68 points from the exam question, while those who spent more than 17 minutes using it received an average of 3.12 points. None of the differences were statistically significant after adjusting the statistical significance level using the Bonferroni correction. This was done in order to avoid the problem of multiple comparisons.

When contrasting the time spent on creating the assignments, we notice that the population that did not spend almost any time on the task is smaller than the population that did not complete any assignments. Thus, there is a population who have attempted to use the tool, but have not completed an assignment due to reasons currently unknown to us. Approximately one third of the population spent more than 17 minutes on creating the assignments.

Similarly, when analyzing the student-written tests for the assignments in more detail, no statistically significant correlation between the exam points from the testing-related exam question and test cases, line coverage, and mutation coverage was identified. This suggests that the quantity or the quality of the test cases that the students implement do not tell much about their exam performance, which also means that the studied context should consider the use of the tool further. It is possible, for example, that the generation of an assignment takes too much focus, and the tests that students create are rather simple. On the other hand, as the testing question in the exam also expects syntactically correct unit tests, which the students do not practice in the tool, it is possible that the exam performance and the tool usage measure different underlying constructs.

5.2 Course material and previous programming experience

The course material also included a worked example on unit testing that the students were expected to follow. It is possible that, as worked examples are a good way for teaching a topic [5], the direct instruction – in a single worked example – worked better than teaching testing with CrowdSorcerer. We must note, however, that in a preliminary investigation in which the tool was isolated from the programming course and shown to a handful of students, our observations indicated that the use of the tool could be a viable approach for easing students into learning testing.

Finally, when analyzing previous programming experience, we note that programming experience has more to do with answering the testing-related exam question than the number of programming assignments created with the tool. On the other hand, creating assignments with the tool might have helped those with no experience in programming to learn testing more than those with some experience. As stated in Table 4, the average score received by the novices is marginally higher than that of those with some programming experience, but the difference is not statistically significant. Interestingly, while novices may be more likely to create an assignment with CrowdSorcerer, they use less time in the tool on average when compared to more experienced programmers. This means that when the more experienced programmers use CrowdSorcerer, they spend more time, possibly creating more thorough or complex assignments.

5.3 Limitations of work

This study comes with a number of limitations, which we address next. First, measuring performance is always hard, and it is possible that the proxies that we chose (exam performance, tool usage,

test quality) are poor approximations of the students' knowledge. Second, the studied exam question was the last question on the exam. While almost everybody answered the exam question, it is possible that some did not have sufficient time to answer the question properly. It is also possible that if students are uncertain of their testing-related skill set, as it is a topic that is not as widely studied on the course, they focus on other exam questions first and try to scramble together an answer at the very end of the exam, which results in lower score. On the other hand, the overall average score from the exam question was 72%, which was close to the average score from the exam overall; it is possible that some of the results are also influenced by a ceiling effect. Third, as we limited our analysis to the population who had attended the exam, there is bound to be selection bias in the data. This is visible also in those who used CrowdSorcerer; the students used the tool voluntarily, and it is possible that we have solely captured the more active population and consequently the use of the tool is secondary. It is possible that this more active population would have performed better regardless. Finally, the way CrowdSorcerer was introduced into the course may have influenced outcomes: the tool was not advertised as a testing learning tool, but it emphasized the benefit of creating a programming assignment and thinking about the process of programming from another perspective, and the students also had a worked example emphasizing unit testing in the material. It is possible that a different kind of focus in the material could have influenced students' focus when working with the tool.

6 CONCLUSIONS

We studied the use of CrowdSorcerer, a tool that can be used for creating and programming assignments with tests. When using it, students first create the problem description, then the model solution, and then the tests for the problem. The tests are written using a simple input-output format, where each input-output test pair provides the input and the expected output. Our hypothesis was that the use of the tool would contribute to students ability to test programs.

The hypothesis was studied using statistical tests between the usage of the tool and a testing-related question in a course exam. We studied whether the number of tests written, the line coverage or the mutation coverage of the students' tests in the student-generated assignments explain students' performance in the testing-related exam question.

To summarize, our answers to the research questions are:

RQ1. Are students who use the tool more likely to answer testing-related exam questions? **Answer:** When analyzing the tool usage and the tendency to answer the testing-related exam question, we found no statistically significant differences between the populations (see Tables 1 and 2).

RQ2. Does the quality of the tests the students create with the tool predict the quality of their answer to testing-related exam questions? **Answer:** None of the correlations (Table 3) between the quality metrics for student-generated assignment tests and exam performance were statistically significant.

RQ3. How does students' previous programming experience influence their willingness to use the tool? **Answer:** Those with no programming experience used the tool marginally more than those with some or a lot of experience (see Table 4). Students with no programming experience also got marginally more points on average in the testing-related question of the course exam than those with at least some experience, but the difference is not statistically significant.

RQ4. Does students' previous programming experience contribute to the quality of written tests? **Answer:** Students with no programming experience got marginally more points on average in the testing-related question of the course exam than those with at least some experience, but the difference is not statistically significant.

Our analysis showed that there was no statistically significant connection between the used metrics and the course outcomes. This highlights a number of issues with learning to write tests, some of which generalize to the broader domain of learning programming. First, it is possible that asking students to generate a full programming assignment is too complex for learning testing, and as such, the methodology should be changed to focus more specifically on tests. It is also possible that the performance metrics chosen here are not optimal, as students were supposed to write syntactically correct tests in the exam, while the tool provided students scaffolding in the form of being able to limit to input-output tests. Similarly, as students were also given a focused worked example on unit testing in the course material, it is possible that it influenced students' behavior and thus reduced the visible effect of the tool.

As a part of our future work, we are looking into elaborating the influence of creating the problem statement and the model solution on students' learning. Do they move students' focus away from the testing task, or do they help students write the tests as they have designed the assignment as well? We are also looking into gamifying the system, as well as diversifying the ways how testing is taught with the tool, for example asking students to create test cases for assignments instead of creating the full assignments.

A APPENDIX - QUESTION: TESTING

You have been given a class **Grading** that reportedly offers the possibility to add course grades, to search for a grade by entering a student ID, and to search for all students who have received a certain grade.

Add a test class **GradingTest** to the code template and write the following unit tests.

- Check that a student with a grade added to the Grading object can be found using the object's **getGrade** method.
- Check that when calling the **getGrade** method for a student ID that has not been added, -1 is returned.
- Check that adds a student with a grade to the Grading object and then verifies that that student is found using the **getStudentsWithGrade** method.
- Check that when there are no students, the method **getStudentsWithGrade** returns an empty list.

Note: Each of the items were graded manually and for each fully working test case, students received 1 point. Deductions were made based on unnecessarily complex code, bad naming of variables and so on.

REFERENCES

[1] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. 2002. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin* 34, 4 (2002), 125–128.

[2] Susan Bergin and Ronan Reilly. 2005. Programming: Factors that Influence Success. In *SIGCSE '05 Proceedings of the 36th SIGCSE technical symposium on Computer science education*. ACM, 411–415. http://eprints.maynoothuniversity.ie/8209/

[3] Barry W. Boehm. 1984. Software Engineering Economics. *IEEE Trans. Softw. Eng.* 10, 1 (Jan. 1984), 4–21. https://doi.org/10.1109/TSE.1984.5010193

[4] David Carrington. 1996. Teaching Software Testing. In *Proceedings of the 2Nd Australasian Conference on Computer Science Education (ACSE '97)*. ACM, New York, NY, USA, 59–64. https://doi.org/10.1145/299359.299369

[5] Ruth C Clark, Frank Nguyen, and John Sweller. 2006. *Efficiency in learning: Evidence-based guidelines to manage cognitive load*. Pfeiffer, John Wiley & Sons.

[6] Benjamin S. Clegg, José Miguel Rojas, and Gordon Fraser. 2017. Teaching Software Testing Concepts Using a Mutation Testing Game. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track (ICSE-SEET '17)*. IEEE Press, Piscataway, NJ, USA, 33–36. https://doi.org/10.1109/ICSE-SEET.2017.1

[7] T. Cowling. 2012. Stages in teaching software testing. In *2012 34th International Conference on Software Engineering (ICSE)*. 1185–1194. https://doi.org/10.1109/ICSE.2012.6227024

[8] Paul Denny, Diana Cukierman, and Jonathan Bhaskar. 2015. Measuring the Effect of Inventing Practice Exercises on Learning in an Introductory Programming Course. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2828959.2828967

[9] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. CodeWrite: Supporting Student-driven Practice of Java. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, NY, USA, 471–476. https://doi.org/10.1145/1953163.1953299

[10] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *J. Educ. Resour. Comput.* 3, 3, Article 1 (Sept. 2003). https://doi.org/10.1145/1029994.1029995

[11] Stephen H. Edwards and Zalia Shams. 2014. Comparing Test Quality Measures for Assessing Student-written Tests. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 354–363. https://doi.org/10.1145/2591062.2591164

[12] Dianne Hagan and Selby Markham. 2000. Does It Help to Have Some Programming Experience Before Beginning a Computing Degree Program?. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '00)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/343048.343063

[13] David Janzen and Hossein Saiedian. 2008. Test-driven Learning in Early Programming Courses. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 532–536. https://doi.org/10.1145/1352135.1352315

[14] Vesa Lappalainen, Jonne Itkonen, Ville Isomöttönen, and Sami Kollanus. 2010. ComTest: a tool to impart TDD and unit testing to introductory level programming. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. ACM, 63–67.

[15] Andrew Luxton-Reilly, Daniel Bertinshaw, Paul Denny, Beryl Plimmer, and Robert Sheehan. 2012. The Impact of Question Generation Activities on Performance. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, USA, 391–396. https://doi.org/10.1145/2157136.2157250

[16] Andrew Luxton-Reilly, Paul Denny, Beryl Plimmer, and Robert Sheehan. 2012. Activities, Affordances and Attitude: How Student-generated Questions Assist Learning. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. ACM, New York, NY, USA, 4–9. https://doi.org/10.1145/2325296.2325302

[17] Will Marrero and Amber Settle. 2005. Testing First: Emphasizing Testing in Early Programming Courses. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. ACM, New York, NY, USA, 4–8. https://doi.org/10.1145/1067445.1067451

[18] Glenford J. Myers. 1979. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

[19] Nea Pirttinen, Vilma Kangas, Irene Nikkarinen, Henrik Nygren, Juho Leinonen, and Arto Hellas. 2018. Crowdsourcing Programming Assignments with CrowdSorcerer. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018)*. ACM, New York, NY, USA, 326–331. https://doi.org/10.1145/3197091.3197117

[20] Nea Pirttinen, Vilma Kangas, Henrik Nygren, Juho Leinonen, and Arto Hellas. 2018. Analysis of Students' Peer Reviews to Crowdsourced Programming Assignments. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. ACM.

[21] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. 2014. No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 469–474. https://doi.org/10.1145/2538862.2538930

[22] Susan Wiedenbeck, Deborah Labelle, and Vennila N. R. Kain. 2004. Factors Affecting Course Outcomes in Introductory Programming. (05 2004), 97–110.

[23] Brenda Cantwell Wilson and Sharon Shrock. 2001. Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. ACM, New York, NY, USA, 184–188. https://doi.org/10.1145/364447.364581