A study on the quality mindedness of students

Steffen Dick[1] Stefan Schulz[2] Christoph Bockisch[3]

Abstract: Awareness of software quality is a skill generally agreed to be very important working in the industry, but we have observed that it receives little attention in the first-year programming education at universities. Besides preparing students for work life, we assume that good knowledge of software quality also helps computer science students to study more successfully. In this paper, we present a method for determining the quality-awareness, based on a diagnostic assignment and a questionnaire. Using the method we establish a baseline measurement in two courses that students typically follow in their first year, showing that quality awareness correlates with good grades. According to the baseline, the level of quality-mindedness of approximately half the students is not satisfactory.

Keywords: code quality awareness, students, learners, survey, evaluation

1 Introduction

It is obvious that writing quality software is very important in practice, improving among others the maintainability of software. As quality attributes, in our context we especially consider readability, following conventions, documentation and sufficient testing. At the same time, it is a known problem that early programming courses at universities do not pay very much attention to software quality because they focus on basic programming concepts and typically only use small exercises. Due to the small size of code to be written and because this code is usually not further developed after it is submitted, the necessity of writing high quality code is not obvious to students.

Our expectation, however, is that early programming education also sets the direction of how students and graduates treat software quality. Furthermore, we expect that, if writing good quality software becomes their second nature, this will positively influence their further success, e.g., because it will improve their capability of working in teams. As part of the Erasmus+ project *QPED* (`https://qped.eu`), we research approaches of strengthening software quality in early programming courses and investigate how they impact the quality awareness in students. In this paper, we report on establishing a baseline measurement of the students' quality awareness and resulting indications that our expectations could be true. In the future, we plan to adapt the syllabus of our first year programming education and repeat these measurements to evaluate changes in quality awareness and overall programming abilities of our students.

[1] Phillipps Universität Marburg, Germany dickst@informatik.uni-marburg.de
[2] Phillipps Universität Marburg, Germany schulzs@informatik.uni-marburg.de
[3] Phillipps Universität Marburg, Germany bockisch@acm.org

doi:10.18420/SEUH2022_12

For that purpose, we devised a study comprised of two parts. Firstly, we developed a diagnostic assignment to assess the students' knowledge of software quality as part of the final exam of an early programming course. Secondly, we developed a survey that lets students in a higher semester rate the importance of different quality criteria. We also used this to gather information on the programming background before their university education.

We elaborate on the results of this survey in the following sections. As one of the main results, we were able to confirm that good knowledge of software quality has a strong positive correlation with success in an early programming course. The questionnaire has revealed that students do already consider software quality rather important—even more so when working in a team—but the quality-awareness still needs to be improved. It has also shown that for most of the students the university courses are the first formal programming education, showing the importance of laying a solid foundation here.

2 Diagnostic Assessment

We wanted to see how knowing about software quality relates to the final grade of early learners. Therefore, we included a task within the final exam about the quality of a given, flawed code snippet. Thereby, the included flaws are no actual bugs, but rather bad style or missing parameter validation. The students were supposed to write a sufficient number of adequate tests and to name and fix a flaw within that code snippet. This was done for the first exam and also for the repeat exam, though we exchanged the code snippet.

For their first exam, the students received a flawed tree-based implementation of the Perrin-sequence which is similar to the Fibonacci sequence that the students were already familiar with. For the repeat exam, we chose a register for citizens which had similar flaws to the implementation of the Perrin-sequence. Over 130 students participated in the first exam and about 60 participated in the repeat exam. Some of these 60 students had already taken the first exam and failed, the others elected to directly participate in the repeat exam. For both exams we used a tool [Dr19] to calculate the item-test-correlation according to Pearson for both sub-tasks. The result is a value between -1 and +1, whereby the +1 is a perfect correlation, 0 is no correlation at all and -1 is a perfect negative correlation. This means students with no quality awareness would have the best grades and vice versa.

For the results of the first exam, we calculated correlations with the final grade of 0.55 for writing the tests, 0.41 for naming and fixing a flaw and 0.61 for the task as a whole. A value above 0.4 is already considered a good correlation [**MoosbruggerKelava2012**]. With smaller sample size of only about 60 students, the item-test-correlation could still be calculated to 0.34 for writing the tests. The second part, naming and fixing a flaw, was skipped by most participants of the repeat exam leading to an item-test-correlation of 0.01 and bringing the correlation of the task as a whole down to 0.25. The high correlation of the first exam, nevertheless, suggests that further research into a causation between quality mindedness and good grades is promising.

3 Surveying the Students

We concluded our evaluation by handing out a survey to participants of a course that is usually situated between the second and third semester. The goal of this survey was to see how much attention students normally pay to software quality when writing source code on their own or within a group. To achieve this, we needed to include some metrics or concepts of software quality within the survey. Going over every metric in existence would have blown up the survey and might have caused less students willing to participate. Because of this, we selected 9 metrics to be included in our survey.

One of the criteria for selecting the metrics is the knowledge of the students. As different cadences of lectures are possible, we had to choose the least common denominator of lectures that we can expect to be completed by the participants. From this, we can expect that students are familiar with the concept of *Magic Numbers* (constants that are not declared as such) and the "Don't Repeat Yourself" principle, which is heavily involved with the metric *Duplicated Code*, so we included both in the survey. The second criterion we considered for selecting the metrics was their naming: We wanted to include those with self-explanatory names so that, even if they hadn't heard of them, the students would be able to have an inkling of what they were about. Metrics such as *Logical Lines of Code*, *Number of Fields*, *Too Large Methods* and *Characters per Line* were included with this reasoning. Our last criterion for selecting a metric was testing. Because testing source code is one of the main pillars of software quality and is taught in both beginner lectures, testing metrics were included. In combination with the second criterion, we chose *Number of Passed Tests*, *Number of Failed Tests* and *Test-Code Coverage*.

Out of 110 students 40 participated in the survey, so the results may be biased, although we do not have any evidence that a specific group of students (e.g., especially good or bad students) stands out in the survey. About 87% of participants identified as male, whereas only 8% identified as female which is below our university's average of about 15-20% female students in computer science. And 5% skipped the question all together.

Of those who participated, 50% were in their second, and about 29% in their third semester. Furthermore, 9% of participants were in their fourth, 8% in their fifth semester and 2% in their tenth semester. This confirms our assumption that the majority of participants were still at the beginning of their university education.

	Compulsory Classes	Optional Classes	Private Means	Industry
None	66%	47%	10.5%	66%
little	5%	10%	18%	10%
some	18%	26%	34%	3%
much	3%	8%	21%	8%
very much	5%	8%	16%	5%

Tab. 1: Participants rating the source of their knowledge about CS before university

We also asked participants to quantify their prior knowledge of computer science before their first semester from various likely sources, the results are shown in Tab. 1. Participants who skipped one or more of these questions are not represented in the table, which is why some columns do not add up to 100%. From the table, we can see that only about 50% of students received formal education on computer science before starting university and that most prior knowledge comes from private means. Almost no student received prior knowledge through working in the industry. So the first formal knowledge half of the students receive comes from university, so it is the foundation for their future careers.

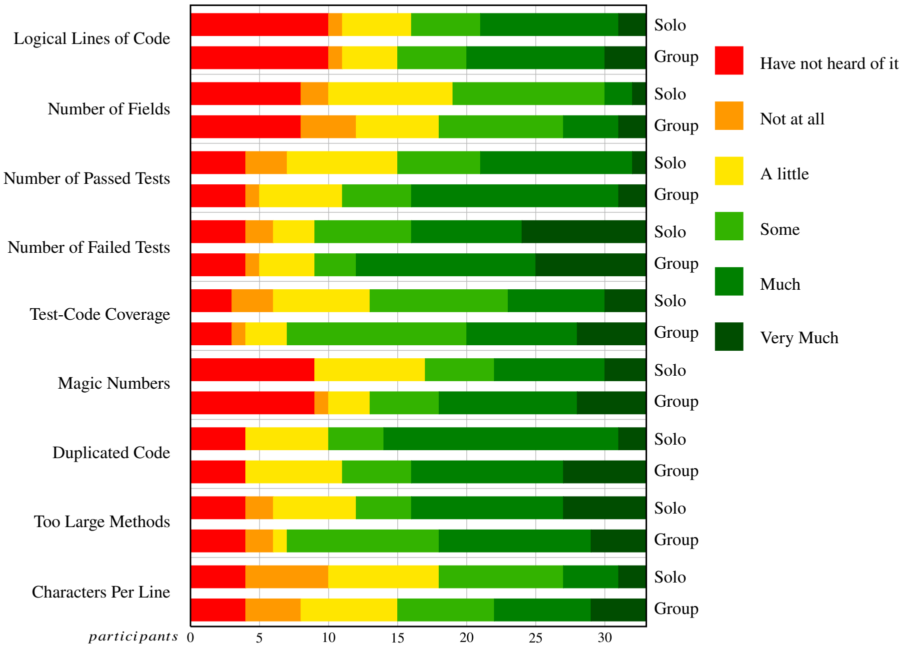

Fig. 1: Participants rating how much attention concepts receive when coding.

Furthermore, we asked the students to rate how much attention the concepts behind the 9 aforementioned metrics receive when they are writing code. They were supposed to rate the concepts when writing code on their own and when within a group of fellow students. The results are shown in Fig. 1. For every concept there are two bars in the chart, with the bottom one being the results when in a group and the top one when writing code on their own. If a student stated that they had not heard of the concept working alone or in a group, the other answer for that concept was changed to *Have not heard of it*. Three answers were demoted this way.

Overall, at least 10% of participants did not know a given metric. Roughly 30% of participants had not even heard about *Magic Numbers*, though it is part of the compulsory curriculum. Only 50% on average stated that they paid *much* attention to the metrics, with *Number of Passed Tests*, *Number of Failed Tests* and *Duplicated Code* receiving the most attention.

4 Related Work

Over the past two decades, a lot of research has been done on the effects of teaching test-driven development concepts on various levels of CS education [DJS08; Ga20; Me20]. Edwards [Ed03] discusses the problem, that many CS students struggle to adopt analysis and comprehension skills, which are crucial for software development. To combat this, they suggest to shift curricula of junior-level courses to a test-first approach and present an automated testing tool, capable of providing helpful feedback to students. In a comparison between courses with and without TDD [Ed04], they observed a reduction of 45% regarding defects in the student's code.

Matthies et al. [MTU17] propose Prof. CI, a novel approach to programming exercises and teaching test-driven development. It leverages modern online courses, designed to teach programming and provides a modern IDE-based tool-chain for the exercises. While the authors did not evaluate the code quality of the student's work, they observed that the students were more likely to write more tests in future projects. An important takeaway is the fact that extra care is required when designing tool-enhanced programming exercises, especially when the exercise requires the students to write tests.

Similar to these works, our goal is to teach better code quality standards to junior-level students by shifting the curriculum of our university's entry level courses. This shift will mainly be achieved by remodelling the mandatory exercises during the semester, using web- and IDE-based tool-support to provide quality-related feedback for the submissions.

5 Conclusion and Future Work

To assess the quality-mindedness of students in their early education, we have performed a study in two courses typically followed in the first and second semester, respectively. This firstly comprised a diagnostic assignment in the final exam which was taken by 130 students on the first exam and 60 students on the repeat exam. An item-test-correlations of 0.6 and 0.25 (in the first and repeat exam, respectively) proves a high correlation between understanding software quality and getting a good grade in the final exam.

Secondly, we made a survey with students in a lab course, most of whom were at the end of their second or third semester. In this survey, 40 students self-appraised how much attention they pay on software quality by rating the importance of nine different quality metrics when developing alone or in a team. In this survey, each quality concept was not known to 10–30% of the students. And on average only half of the students paid much attention to the quality concepts. This shows that it is a good idea to increase the efforts to teach early learners about software quality to improve their quality-mindedness.

In the future, we will update the syllabus of our introductory course on object-oriented programming to strengthen the topic of software quality and to make students more aware

of the importance of paying attention to software quality. We also plan to repeat the study presented in this paper with students who followed the updated course to verify our assumption that by changing the syllabus we can increase the quality-mindedness of students. Furthermore, we will investigate if the changed syllabus will improve the achievements in the exam regarding software quality, and whether an increased quality-mindedness also leads to better overall grades. If this can be shown, it would allow us to conclude that quality-mindedness is not only correlated with good grades but also a reason for them.

Acknowledgements

This work is partly funded by the Erasmus+ project *Quality-focussed Programming Education (QPED)*, 2020-1-NL01-KA203-064626.

References

[DJS08] Desai, C.; Janzen, D.; Savage, K.: A survey of evidence for test-driven development in academia. ACM SIGCSE Bulletin 40/2, 2008.

[Dr19] Dreyer, T.: Automatische Gütebewertung von Informatikklausuren, Bachelor Thesis, Sept. 2019.

[Ed03] Edwards, S. H.: Rethinking computer science education from a test-first perspective. In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2003.

[Ed04] Edwards, S. H.: 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. 2004.

[Ga20] Garousi, V.; Rainer, A.; Lauvås Jr, P.; Arcuri, A.: Software-testing education: A systematic literature mapping. Journal of Systems and Software 165/, 2020.

[Me20] Melo, S. M.; Moreira, V. X.; Paschoal, L. N.; Souza, S. R.: Testing Education: A Survey on a Global Scale. In: Proceedings of the 34th Brazilian Symposium on Software Engineering. 2020.

[MTU17] Matthies, C.; Treffer, A.; Uflacker, M.: Prof. CI: Employing continuous integration services and Github workflows to teach test-driven development. In: 2017 IEEE Frontiers in Education Conference (FIE). IEEE, 2017.