Pragmatic software testing education

Aniche, Maurício; Hermans, Felienne; van Deursen, Arie

DOI
10.1145/3287324.3287461

Publication date
2019

Document Version
Accepted author manuscript

Published in
SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education

Pragmatic Software Testing Education

Maurício Aniche
Delft University of Technology
The Netherlands
m.f.aniche@tudelft.nl

Felienne Hermans
Delft University of Technology
The Netherlands
f.f.j.hermans@tudelft.nl

Arie van Deursen
Delft University of Technology
The Netherlands
arie.vandeursen@tudelft.nl

ABSTRACT

Software testing is an important topic in software engineering education, and yet highly challenging from an educational perspective: students are required to learn several testing techniques, to be able to distinguish the right technique to apply, to evaluate the quality of their test suites, and to write maintainable test code. In this paper, we describe how we have been adding a pragmatic perspective to our software testing course, and explore students' common mistakes, hard topics to learn, favourite learning activities, and challenges they face. To that aim, we analyze the feedback reports that our team of Teaching Assistants gave to the 230 students of our 2016-2017 software testing course at Delft University of Technology. We also survey 84 students and seven of our teaching assistants on their perceptions. Our results help educators not only to propose pragmatic software testing courses in their faculties, but also bring understanding on the challenges that software testing students face when taking software testing courses.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Software and its engineering** → *Software verification and validation*;

KEYWORDS

software testing education, software engineering education, computer science education.

ACM Reference Format:
Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *SIGCSE '19: 50th ACM Technical Symposium on Computer Science Education, February 27–March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287461

1 INTRODUCTION

Every software developer should be aware of the (high) impact that malfunctioning software can have in our society. We have seen huge losses in the financial market [30], and even researchers withdrawing their papers [33]; all of them caused by software bugs. Making sure software works is maybe the greatest responsibility of a software developer. Luckily, over the years, software testing moved away from being considered the activity that 'less skilled' software engineers do to one of the most important skills an engineer should have.

The act of inspecting large and complex code bases to find bugs is not a trivial task in the real world: engineers need to have a broad understanding of different practices that vary from simple manual exploratory testing, where a human tries to find bugs manually by interacting with the system, to advanced bleeding-edge testing techniques, such as automated testing and automated test generation, where engineers program machines to test their system.

Companies such as Facebook [12], Google [41], and Microsoft [35] take testing seriously and require their engineers to master such techniques. Surveys have shown that developers understand the importance of testing-related training [15] and yet many of them still lack formal testing education [6, 34].

Indeed, educating a student in the art of software testing is challenging, for both students and educators. From the educator's perspective, it is hard to keep a testing course up-to-date with the novelties of the field as well as to come up with exercises that are realistic [14]. Due to the importance of the topic, educators have been experimenting with the introduction of testing earlier in Computer Science programs [17, 19–21, 23, 27], introducing a test-first approach in CS courses [9, 10, 22], developing tools focused on software testing education [11, 38], and proposing more complete postgraduate courses focused on testing [39]. Educators also face the fact that some testing topics are not conceptually straightforward, not easy to demonstrate and generalize, and are not all available in a single textbook [40].

This paper has a twofold goal. First, to present how we have been teaching pragmatic software testing to the first year CS students at Delft University of Technology. Second, we explore students' common mistakes, hard topics to learn, favourite learning activities, and challenges they face when learning pragmatic software testing.

To this aim, we analyzed the 1,993 quotes from the feedback report that we, as teachers and teaching assistants, gave to each of the 230 students of the 2017 edition of the Software Quality and Testing course, which is taught at the first year of our Computer Science bachelor. In addition, we performed a survey with 84 students, which we augmented by also surveying seven of our TAs.

The main contributions of this paper are:

- A proposal for a pragmatic software testing course based on nine key principles that can be taught for computer science students, including building a test mindset and interaction with practitioners (Section 3).
- An empirical analysis of the students' most common mistakes (Section 6.1), their perceptions on the most difficult topics in software testing (Section 6.2), and the importance of different teaching activities (Section 6.3) when learning pragmatic software testing.

SIGCSE'19, February 27–March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/2…$15.00
https://doi.org/10.1145/3287324.3287461

2 RELATED WORK

Software Testing is an important part of any Software Engineering program [2, 8, 26, 42], and by itself poses *several other challenges* to educators. Unfortunately, the topic still does not receive its deserved attention in several CS programs. Wong [42] argues that many engineers are not well trained in software testing because most CS programs offer ST as an elective course. Clarke et al. [8] also points to the fact that due to the large number of topics to be covered in a Software Engineering program, little attention is given to Software Testing. Astigarraga et al. [2] show that most CS programs tend to emphasize development at the expense of testing as a formal engineering discipline. Lemos et al. [26] show that software testing education can improve code reliability in terms of correctness; however, authors also argue that university instructors tend to lack the same knowledge that would help students increase their programming skills toward more reliable code.

Educators have been suggesting different approaches on how to introduce testing in a CS curriculum: from students submitting their assignments together with test plans or sets [16, 17, 21], performing black-box testing in a software seeded with errors [21, 24, 31], students testing each others' programs [36], to suggesting students to use a test-first approach at the very beginning of the program [9, 10, 22, 27]. Many of these authors even suggest that tests should be incorporated to the Computer Science and Software Engineering curricula, not only as an elective discipline, but throughout the curriculum. More specifically, Jones [23] suggests that students need to see the practice of software testing as part of the educational experience and that each core course in the curriculum should impart one or more testing experiences.

In addition, educators have proposed tools that are solely focused on software testing education. Elbaum et al. [11] propose *BugHunt*. BugHunt is a tool that contains four different lessons on software testing (terminology, black box, white box, efficiency in testing). 79% of the students in their experiment agreed that BugHunt added significant value to the material presented in the lecture(s) on software testing, and 61% of the students agreed that BugHunt could replace the classes on testing. Spacco and Pugh propose *Marmoset* [38], a tool to help incentivize students to test their software. Marmoset's innovative element is that if a submission passes all of the public test cases, then students are given the opportunity to test their code against a test suite that is not publicly disclosed.

3 PRAGMATIC SOFTWARE TESTING EDUCATION

The *Software Testing and Quality Engineering* at Delft University of Technology is a course that covers several different aspects of software testing, ranging from topics in the ISTQB industry certification [5] to software testing automation, as well as the future of testing by means of selected research papers.

The course is currently a compulsory part of the 4th quarter of the first year in the Computer Science bachelor. The course corresponds to 5 ECTS (140 hours). Students have two lectures of 1.5 hours plus 4 hours of labwork a week. As a pre-requisite, students should have at least basic knowledge on Java programming language.

The teaching team is currently composed of two teachers and teaching assistants (TAs). The number of TAs vary as our university has a policy of 1 TA per 30 students. Teachers are responsible for the course design, lectures, creating and assessing multiple choice exams, and they have the overall responsibility of the course. TAs are responsible for helping students, grading all labwork deliverables, and for giving concrete and specific feedback on what students can improve.

Learning goals. At the end of the course, students (1) are able to create unit, integration, and system tests using current existing tools (*e.g.*, JUnit, Mockito) that successfully test complex software systems, (2) are able to derive test cases that deal with exceptional, corner, and bad weather cases by performing several different techniques (*i.e.*, boundary analysis, state-based testing, decision tables), (3) are able to measure and reflect on the effectiveness of the developed test suites by means of different test adequacy metrics (*e.g.*, line and branch code coverage, MC/DC), (4) are able to reflect on limitations of current testing techniques, when and when not to apply them in a given context, and to design testable software systems, (5) Participants are able to write maintainable test code by avoiding well-known test code smells (*e.g.*, Assertion Roulette, Slow or Obscure Tests).

Program. The course covers software quality attributes, maintainability and testability, manual and exploratory testing, automated testing, devops, test adequacy, model-based testing, state-based testing, decision tables, reviews and inspections, design-by-contract, embedded system testing, test-driven design, unit versus integration testing, mocks and stubs. More specifically:

- Week 1: Introduction to software testing, fault vs failure, principles of testing, (un)decidabilitily, introduction to JUnit, introduction to labwork.
- Week 2: Life cycle, validation vs verification, V-model, code reviews. Functional testing, partition testing, boundary testing, and domain testing.
- Week 3: Structural testing, adequacy criteria, code coverage. Unit vs integration vs system testing, mock objects, and test-driven development.
- Week 4: State-based testing, model-based testing, and decision tables.
- Week 5: Test code quality, test code smells. Design for testability. Design-by-contracts.
- Week 6: Security testing. Search-based software testing.
- Week 7: Guest lectures from industry.

Key elements. To achieve a pragmatic software testing course, we have devised and currently follow some key elements:

Theory applied in the lecture. We put our efforts into developing lectures where students can see theory being applied to practice. Our lectures often have the following structure: we present a (buggy) code implementation (initially on slides, and later in the IDE), we discuss where the bug is, we explore, at a conceptual level, a systematic approach to detect the bug, we apply the approach into a set of concrete examples. In other words, we do not only focus on explaining abstract ideas, but on concretely showing how to apply them on different real world problems, using real-world tools, like JUnit, Mockito, and Cucumber.

Real-world pragmatic discussions. Software testing is a challenging activity to be done in practice. This means that developers often make trade-offs in deciding what and how much to test. Engineering questions that arise when complex software systems are being tested, such as "how much should I test?", "how should I test a mobile application that communicates with a web server?", and "should I use mocks to test this application?" are often discussed in classroom so that students see how to extrapolate from our often small exercises to their future real lifes as developers.

Build a testing mindset. Software testing is not seen as an important task by many students. A software testing course should inspire students to think about testing whenever they implement any piece of code. In our testing course, we aim to achieve such a testing mindset by (1) showing how testing can be a creative activity, requiring strong developers, by means of several live coding sessions and rich pragmatic discussions, (2) demonstrating not only the usefulness of any testing technique we teach, but also how they are applied, as well as what trade-offs such techniques have in the real–world, (3) bringing guest lecturers who talk about the importance of software testing for their companies.

Software testing automation. The software engineering industry has long been advocating the automation of any software testing activity [12, 35, 41]. However, some software testing courses still focus on writing test case specifications solely as documents, and do not discuss how to automate them. In our course, to all the theoretical and systematic test design techniques we present, from functional testing to structural testing, from unit to system-level tests, students later write them in a form of an automated test. Mastering tools such as JUnit and Mockito, standard tools for test automation in Java, is a clear learning goal of our course. The importance of automation also strongly appears in our labwork, which we discuss next.

A hands-on labwork. We see the labwork as an important learning method. In our course, by means of a practical labwork assignment, students apply a selection of techniques to a 3k lines of code game written in Java, namely, JPacMan. The labwork contains a set of 50 exercises in which students are able to exercise all the techniques we teach. It is important to notice that students not only generate test cases on the paper, but also automate them. A great amount of their work is in actually producing automated JUnit test cases.

In the following, we present the main deliverables of our labwork. The complete assignment can be found in our online appendix [1].

- *Part 0 (Pre-requisites).* Clone the project from Github, configure the project in your IDE, write your first JUnit test, run coverage analysis.
- *Part 1.* Write a smoke test, functional black-box testing, boundary tests, reflect on test understandability and best practices.
- *Part 2.* White-box testing, mock objects, calculate code coverage and apply structural testing, use decision tables for complex scenarios, reflect on how to reduce test complexity and how to avoid flaky tests.
- *Part 3.* Apply state-based testing, test reusability, refactor and reflect on test smells.

Test code quality matters. Due to the importance of automated testing activities, software testers will deal with large test codebases. Empirical research has indeed shown that test code smells often happen in software systems, and that their presence has a strong negative impact on the maintainability of the affected classes [3]. We often reinforce the importance of refactoring test code and make sure they are free of smells. To any test code we write during live coding sessions, we make sure that they are as free of smells as possible. Test smell catalogues such as the ones proposed by Meszaros [32] are deeply discussed in a dedicated lecture.

Design systems for testability. Designing software in such a way that it eases testability is a common practice among practitioners [13, 18, 29]. This requires us to not only discuss software testing in our course, but software architecture and design principles of testable software systems, such as dependency inversion [28], observability and controllability, in an entire dedicated lecture for the topic. Questions like "Do I need to test this behavior via an unit or a system test?", "How can I test my mobile application?" are extensively discussed not only through the eyes of software testing, but also to the eyes of software design.

Mixture of pragmatic and theoretical books. The two books we use as textbooks in the course are the *"Foundations of software testing: ISTQB certification"* [5], which gives students a solid foundation about testing theory, and the *"Pragmatic Unit Testing in Java 8 with JUnit"* [25], which gives students concrete and practical examples on how to use testing tools, like JUnit. We believe both complement each other and both are important for students who will soon become a software tester.

Interaction with practitioners. We strongly encourage their interaction with practitioners throughout our course. Having guest lectures from industry practitioners helps us to show the pragmatic side of software testing. Guests focus their lectures on how they apply software testing at their companies, tools they use, their pros and cons, and on the mistakes and challenges they face. In the 2017 edition, we also experimented with Ask-Me-Anything (AMA) sessions, where we called experts from all over the world via Skype and students had 15 minutes to ask any software-testing related questions.

Grading. We currently use the following formula to grade our students: `0.25 * labwork + 0.75 * exam`. The labwork (as we explain below) is composed of 4 deliverables, each graded by our TAs in a range of [0..10]. We later average the grades of four deliverables, which compose the labwork component of the grade. At the end of the course, we propose a 40-question multiple choice exam. Students may take a resit 6 weeks later if they did not pass in the first time. We also offer an optional midterm exam for students who want to practice beforehand.

4 RESEARCH METHODOLOGY

The *goal* of this study is to provide a better understanding of the difficulties and challenges that students face when learning pragmatic software testing.

To that aim, we analyze the data from 230 students of the 2016-2017 edition of our software testing course. We propose three research questions:

RQ$_1$: What common mistakes do students make when learning software testing?

RQ$_2$: Which software testing topics do students find hardest to learn?

RQ$_3$: Which teaching methods do students find most helpful?

To answer our research questions, we collect and analyze data from three different sources: the feedback reports that TAs give to students throughout the course, a survey with students, and a survey with the TAs, both performed after the course. We characterize the participants in Section 5. In the following, we detail the three parts of our methodology.

Manual content analysis on the feedback. As we explain in Section 3, students work on and produce four deliverables during the course. After each deliverable, our team of TAs manually reads students' reports, source code, and tests, and with the help of a rubric, provides them with rich qualitative feedback.

This feedback usually contains several quotes that touch on a mix of different topics, such as mistakes they made in the exercises, tips on how to improve their existing work, issues on the written report, and even compliments for their good work. The language of such feedback reports is usually informal, as we do not give constraints to TAs on how the feedback should be.

We analyze the content of all feedback reports. To that aim, we first filter out any feedback that is not directly related to software testing (*e.g.*, comments on exercises that were not done, or compliments). We then follow an iterative process, derived from standard qualitative data analysis procedures [37]: (1) we assign a code for each quote in the feedback; the code summarizes the essence of the quote, (2) if a quote does not belong to any existing codes, we introduce a new code, (3) each quote has just a single code; if a quote tackles two different problems, we split the original quote into two quotes, (4) to assign the correct code to a quote, we used our knowledge of the testing course, labwork, and the existing rubrics. We assigned 40 different codes to a total of 1,993 quotes. As a next step, we started an iterative merging process to derive the final themes, by grouping similar codes into higher-level themes, *e.g.*, the theme "maintainability of test code" contains quotes from the "test quality", and "test duplication" codes. We ended up with eight themes that we present in the Results (Section 6).

Survey with students. With the goal of capturing their perceptions on learning software testing, we asked students to answer a questionnaire that contained both open and closed questions at the end of the course.

The survey contains a total of 18 questions, none of which are required. The two closed questions of the survey asked students about the difficulty of learning and putting into practice the concepts and techniques we taught, and about the importance of the different activities we used throughout the course. In these questions, students had to choose from a five point Likert-scale, ranging from strongly disagree to strongly agree (see Figures 2 and 3). The open questions were mostly focused on understanding the students' main challenges, difficulties, and suggestions of improvements for our testing course. We apply qualitative techniques to analyze the results of each open question individually, similarly to our analysis of the feedback reports. The full survey as well the full code book can be found in our online appendix [1].

We did not make answering the survey compulsory for the students. We received 84 complete answers out of the 230 students.

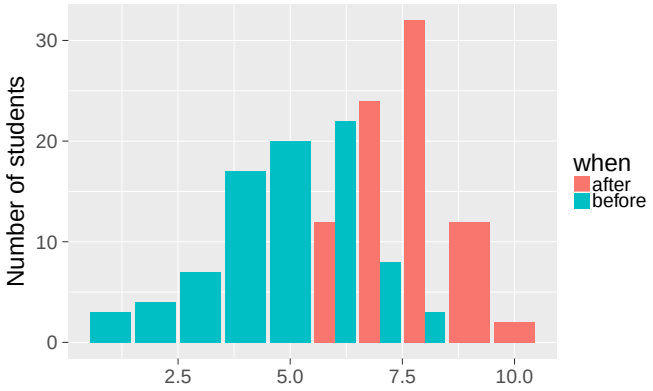

Figure 1: Histogram of the students' (X axis) perceptions in terms of their knowledge (Y axis) on software testing before and after the course. Scale: from 1 to 10.

Survey with Teaching Assistants. Our TAs support students throughout the course, by answering their questions, supporting their work during the lab, and by grading their assignments. As a consequence of such intense contact with students, TAs obtain a good perspective on the challenges of teaching software testing.

We also performed a similar survey with TAs, focusing on what they perceive as challenges for students. The survey contained the same two closed questions from the students' survey (challenges when applying software testing, and the importance of the different activities). In the open questions, we focused on asking about the common mistakes students do during the lab, as well as their perceptions on the challenges that students face.

We shared the survey internally at the end of our course. We also did not make answering the survey compulsory for TAs. At the end, we received 7 complete answers out of the 10 TAs.

5 CHARACTERIZATION OF THE PARTICIPANTS

Students. 66 students identify themselves as male, 8 as female, and 10 preferred not to answer. 89.3% of the students are between 18 to 24 years, five are between 25 and 34, and four are 17 or younger. Only three students were international students. In terms of Java knowledge, in a scale from 1 to 10, 9.5% of students consider their knowledge between 9 and 10, and 72% of them consider themselves between 7 and 8. Only 4 students consider themselves 5 or below.

Thanks to the introduction to JUnit that students receive during their very first course on programming, most of them already had some knowledge on software testing prior to our course. In fact, as we show in Figure 1, before the course starts, in a scale from 1 to 10, 39% of them consider themselves between 6 and 8, 44% between 4 and 5, and only 16% between 1 and 3. No student considered herself a 9 or 10. Students considered that their knowledge increased after the course. All of them considered their knowledge after the course as 6 or greater; 39% of them ranked themselves with a 8, and 14.6% with a 9. Two students ranked themselves with a 10.

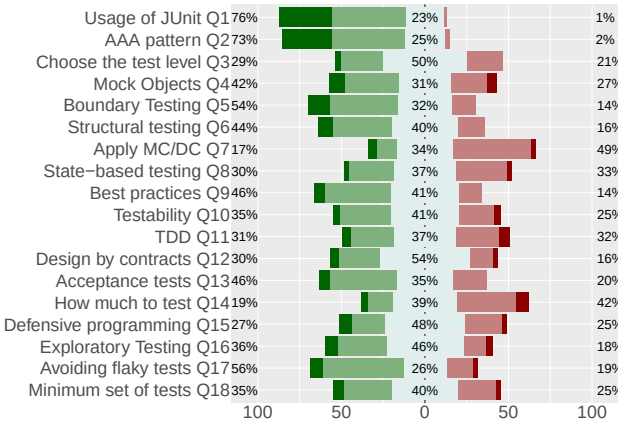

Figure 2: Students' perception on the difficulty of each testing topic. Scale: very easy, easy, neutral, hard, very hard. The full questionnaire can be found in our appendix [1].

Teaching Assistants. All TAs are between 18 and 24 years-old, one of them being female. They all ranked their Java knowledge between 8 and 10, and software testing knowledge between 7 and 8. Four of them are TAs for the first time in our software course; the other three TAs are performing this role for the third year in a row.

6 RESULTS

6.1 RQ_1: What common mistakes do students make when learning software testing?

We characterize the labwork feedback in eight different themes (ordered by their frequency): *test coverage*, *maintainability of test code*, *understanding testing concepts*, *boundary testing*, *state-based testing*, *assertions*, *mock objects*, and *tools*.

Test coverage (416 times, 20.87%). Students commonly either miss tests, *i.e.*, they do not provide all the expected tests for a given piece of code, or they write tests that are not totally correct, *e.g.*, the test does not actually test the piece of code, or the test exercises the wrong class. In addition, we also observed cases (14) where the student actually "overtested" (*i.e.*, wrote tests for more cases than required).

Maintainability of test code (407 times, 20.42%). Students often need advice on how to write maintainable test code. More specifically, test quality advices in general, such as better naming and excessive complexity (247), code duplication and lack of reusability (69), tests that could be split in two (31), better usage of test cleanup features, such as JUnit's Before and After (47).

Understanding testing concepts (306 times, 15.35%). Students provide incomplete answers or have difficulties when it comes to questions that involve testing concepts and ideas, such as what flaky tests are about, advantages and disadvantages of unit and system tests, and the importance of removing test smells.

Boundary testing (258 times, 12.95%). Students often miss all the tests required to cover a boundary (142). As we also ask them to first build a decision table and then derive the tests, we also see that they often miss elements in the table (50) and generate tables that are not fully correct (46).

State-based testing (247 times, 12.39%). When it comes to state-based testing, students often miss or create wrong states or events (56) and transitions (72), or develop non-clear or not legible state machines (68).

Assertions (158 times, 7.93%). Most feedback related to assertions focus on missing assertions, *i.e.*, the student forgot to assert one or more expected result, and on assertions that are wrong or should not exist in that test.

Mock Objects (117 times, 5.87%). Students required some feedback on how to use mock objects. More specifically, on how to properly verify interactions with mock objects (*i.e.*, Mockito's 'verify' method) and to explain when one should mock an object.

Tools (84 times, 4.21%). Students sometimes do not use the tools properly. More specifically to our course, students commonly use JUnit 4 features instead of JUnit 5, do not correctly use AssertJ's fluent API, and make wrong use of Cucumber features.

TAs perspective. Overall, the observations of TAs match with what we observed in the labwork analysis. In terms of testing best practices, TAs mentioned to help students in writing maintainable test code. According to one TA, students often write tests that contain unnecessary code and weird interactions with the class under test. In addition, according to one TA, students do not clearly see how to reuse test code. Another TA mentioned that a common question is on how to properly test exceptions. Finally, a TA also observed that students often write tests that actually do not exercise any production code (in this case, JUnit still shows a green bar, giving a false impression of success to the student).

6.2 RQ_2: Which software testing topics do students find hardest to learn?

In Figure 2, we show, based on the survey data, how students and TAs perceive the difficulty of each of the topics we teach.

Most students consider using the JUnit framework (Q1) as well as to think about the Act-Arrange-Assert pattern that composes any unit test (Q2) easy to learn. In fact, 76% and 73% of students consider it easy or very easy to learn JUnit and to use the AAA pattern, respectively. These perceptions are also shared by TAs, and matches with RQ_1 results, as the number of feedback related to bad tool usage is small (4.21%).

Interestingly, applying MC/DC (Modified Condition/Decision Coverage) [7] criteria to test complicated conditions (Q7) was considered hard or very hard by 49% of the students; this is the hardest topic among all of them. However, it seems that other coverage criteria are easier to learn, as only 16% of students considered structural testing hard (Q6).

Applying software testing in a pragmatic way, as expected, was considered hard for students. Deciding how much testing is enough (Q14) is also considered a hard topic by 42% of students (the second most hard topic). TAs agree and even perceive this topic harder than the students. This result also matches with our findings on RQ_1, where test coverage is the most prominent topic in feedback. In addition, writing the mininum set of tests that gives confidence (Q18) is considered hard for 25% of students and neutral for 40%. Choosing the right level of testing (*e.g.*, unit, integration, or system tests) is not considered easy to all of them; 29% consider it easy,

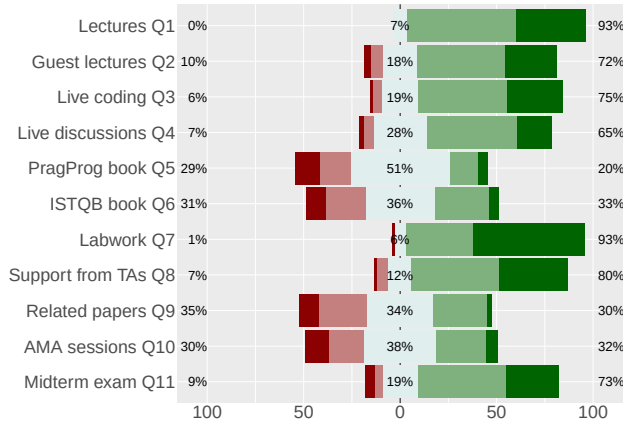

Figure 3: Importance of different activities in software testing learning. Scale=strongly disagree, disagree, neutral, agree, strongly agree. The full questionnaire can be found in our appendix [1].

50% of the students are neutral, and 21% perceive it as a hard topic (Q3). Not a single TA perceived this topic as easy. We believe these findings highlight even more the importance of discussing even more the pragmatic side of software testing.

When it comes to testing code best practices, students had a contradicting perceptions. The usage of mocks to simulate a dependency (Q4) and writing fast, reproducible, and non-flaky tests (Q17) were considered easy topics to be learned by 42% and 56% of students, respectively. TAs agree that students learn these topics with less difficulties. However, when it comes to following testing best practices (Q9), 46% of students perceive it as an easy topic, while 71% of TAs perceive it as a hard topic for students. The students' perceptions also contradicts the results of RQ_1, where we observe a large number of feedback focused on best practices in their assignments.

Finally, testability seems less challenging for students than for TAs. While students perceive optimizing code for testability (Q10) as just somewhat challenging (35% find it easy, 41% are neutral, and 25% find it hard), 67% of TAs belive that testability is a hard topic for students. As we conjecture that TAs have a better understanding of testability than the students, these findings suggest that the students are not sufficiently aware of the difficulty of testability.

6.3 RQ_3: Which teaching methods do students find most helpful?

In Figure 3, we show how students perceive the importance of each learning activity we have in our software testing course.

Students perceive activities that involve practitioners as highly important. More specifically, guest lectures from industry (Q2) were considered important by 72% of participants. The Ask-me-Anything sessions (Q10), on the other hand, was considered important by only 32% of participants; 38% are neutral, and 30% do not consider them important.

Moreover, different interactions during the lecture are also considered important for students. Teachers performing live code (Q3) and discussions and interactions during the lecture (Q4) are considered important by 75% and 65% of students, respectively. We conjecture that discussions and live coding are moments in which students have the opportunity to discuss the topics they consider hard, such as how much testing is enough, which test level to use, and test code best practices (as seen in RQ_1 and RQ_2).

On the other hand, the two books we use as textbooks in the course are not considered fundamental for students. More specifically, 31% of students find the ISTQB [5] not important and 36% are neutral (Q6), whereas 29% of them find the PragProg [25] not important and 51% are neutral (Q5). Reading related papers (Q9) is also considered not important for 35% of them.

6.4 Limitations of our study

The qualitative analysis of the open questions in the survey was manually conducted by the first author of this paper. The analysis, therefore, could be biased towards the views of the authors. To mitigate the threat, we make all the data available for inspection in our online appendix [1],

TAs were responsible for giving feedback to students throughout the study. Although we instruct all TAs on how to grade and what kind of feedback to give (they all follow the same rubrics), different TAs have different personalities. In practice, we observed that some TAs provided more feedback than other TAs. While we believe this could have little impact on the percentages of each theme in RQ_1, we do not expect any other theme to emerge,

In terms of generalizability, although we analyzed the behavior of 230 students, we do not claim that our results are complete and/or generalizable. Furthermore, most students were dutch (we only had 3 international students answering our survey), which may introduce cultural bias to our results. We urge researchers to perform replications of this study in different countries and universities.

7 CONCLUSIONS

Software testing is a vital discipline in any Software Engineering curriculum. However, the topic poses several challenges to educators and to students. In this paper, we proposed a pragmatic software testing curriculum and explored students' common mistakes, hard topics to learn, favourite learning activities, important learning outcomes, and challenges they face when studying software testing.

Researchers and educators agree that software testing education is fundamental not only to industry, but also to research. We hope this paper helps the community to improve even more the quality of their software testing courses. As Bertolino [4] states in her paper on the achievements, challenges, and dreams on software testing research: *"While it is research that can advance the state of the art, it is only by awareness and adoption of those results by the next-coming generation of testers that we can also advance the state of practice. Education must be continuing, to keep the pace with the advances in testing technology".*

ACKNOWLEDGMENTS

We thank all the students and teaching assistants that followed our course in the last years.

REFERENCES

[1] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2018. Pragmatic Software Testing Education: Appendix. (2018). https://doi.org/10.5281/zenodo.1459654.

[2] Tara Astigarraga, Eli M Dow, Christina Lara, Richard Prewitt, and Maria R Ward. 2010. The emerging role of software testing in curricula. In *Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments, 2010 IEEE*. IEEE, 1–26.

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.

[4] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*. IEEE Computer Society, 85–103.

[5] Rex Black, Erik Van Veenendaal, and Dorothy Graham. 2012. *Foundations of software testing: ISTQB certification*. Cengage Learning.

[6] FT Chan, TH Tse, WH Tang, and TY Chen. 2005. Software testing education and training in Hong Kong. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*. IEEE, 313–316.

[7] John Joseph Chilenski and Steven P Miller. 1994. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 9, 5 (1994), 193–200.

[8] Peter J Clarke, Debra Davis, Tariq M King, Jairo Pava, and Edward L Jones. 2014. Integrating testing into software engineering courses supported by a collaborative learning environment. *ACM Transactions on Computing Education (TOCE)* 14, 3 (2014), 18.

[9] Stephen H Edwards. 2003. 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*. ACM, 148–155.

[10] Stephen H Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin* 36, 1 (2004), 26–30.

[11] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. 2007. Bug hunt: Making early software testing lessons engaging and affordable. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 688–697.

[12] Facebook. [n. d.]. Building and Testing at Facebook. https://www.facebook.com/notes/facebook-engineering/building-and-testing-at-facebook/10151004157328920/. ([n. d.]). Last visited in October, 2017.

[13] Steve Freeman and Nat Pryce. 2009. *Growing object-oriented software, guided by tests*. Pearson Education.

[14] Vahid Garousi and Aditya Mathur. 2010. Current state of the software testing education in north american academia and some recommendations for the new educators. In *Software Engineering Education and Training (CSEE&T), 2010 23rd IEEE Conference on*. IEEE, 89–96.

[15] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.

[16] Judith L Gersting. 1994. A software engineering "frosting" on a traditional CS-1 course. In *ACM SIGCSE Bulletin*, Vol. 26. ACM, 233–237.

[17] Michael H Goldwasser. 2002. A gimmick to integrate software testing throughout the curriculum. In *ACM SIGCSE Bulletin*, Vol. 34. ACM, 271–275.

[18] Misko Hevery. 2008. Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process.. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 747–748.

[19] Thomas B Hilburn. 1996. Software engineering-from the beginning. In *Software Engineering Education, 1996. Proceedings., Ninth Conference on*. IEEE, 29–39.

[20] Thomas B Hilburn and Massood Townhidnejad. 2000. Software quality: a curriculum postscript?. In *ACM SIGCSE Bulletin*, Vol. 32. ACM, 167–171.

[21] Ursula Jackson, Bill Z Manaris, and Renée A McCauley. 1997. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. In *ACM SIGCSE Bulletin*, Vol. 29. ACM, 360–364.

[22] David Janzen and Hossein Saiedian. 2008. Test-driven learning in early programming courses. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 532–536.

[23] Edward L Jones. 2001. An experiential approach to incorporating software testing into the computer science curriculum. In *Frontiers in Education Conference, 2001. 31st Annual*, Vol. 2. IEEE, F3D–7.

[24] Edward L Jones. 2001. Integrating testing into the curriculum—arsenic in small doses. *ACM SIGCSE Bulletin* 33, 1 (2001), 337–341.

[25] Jeff Langr, Andy Hunt, and Dave Thomas. 2015. *Pragmatic unit testing in Java 8 with JUnit*. The Pragmatic Bookshelf.

[26] Otávio Augusto Lazzarini Lemos, Fábio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. 2017. The impact of Software Testing education on code reliability: An empirical assessment. *Journal of Systems and Software* (2017).

[27] Will Marrero and Amber Settle. 2005. Testing first: emphasizing testing in early programming courses. In *ACM SIGCSE Bulletin*, Vol. 37. ACM, 4–8.

[28] Robert C Martin. 2002. *Agile software development: principles, patterns, and practices*. Prentice Hall.

[29] Robert C Martin. 2017. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press.

[30] Scott Matteson. [n. d.]. Report: Software failure caused 1.7 trillion in financial losses in 2017. https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/. ([n. d.]).

[31] Renée McCauley and Ursula Jackson. 1999. Teaching software engineering early: experiences and results. *ACM SIGCSE Bulletin* 31, 2 (1999), 86–91.

[32] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[33] G. Miller. [n. d.]. A Scientist's Nightmare: Software Problem Leads to Five Retractions. http://science.sciencemag.org/content/314/5807/1856.full. ([n. d.]). Last visited in October, 2017.

[34] SP Ng, Tafline Murnane, Karl Reed, D Grant, and TY Chen. 2004. A preliminary survey on software testing practices in Australia. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. IEEE, 116–125.

[35] Alan Page, Ken Johnston, and Bj Rollison. 2008. *How we test software at Microsoft*. Microsoft Press.

[36] James Robergé and Candice Suriano. 1994. Using laboratories to teach software engineering principles in the introductory computer science curriculum. In *ACM SIGCSE Bulletin*, Vol. 26. ACM, 106–110.

[37] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage.

[38] Jaime Spacco and William Pugh. 2006. Helping students appreciate test-driven development (TDD). In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 907–913.

[39] Muhammad Dhiauddin Mohamed Suffian, Suhaimi Ibrahim, and Mohamed Redzuan Abdullah. 2014. A proposal of postgraduate programme for software testing specialization. In *Software Engineering Conference (MySEC), 2014 8th Malaysian*. IEEE, 342–347.

[40] Joseph Timoney, Stephen Brown, and Deshi Ye. 2008. Experiences in software testing education: some observations from an international cooperation. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*. IEEE, 2686–2691.

[41] James A Whittaker, Jason Arbon, and Jeff Carollo. 2012. *How Google tests software*. Addison-Wesley.

[42] Eric Wong. 2012. Improving the state of undergraduate software testing education. In *American Society for Engineering Education*. American Society for Engineering Education.