How do students test software units?

How do students test software units?

Part one: Their natural attitude diagnosed

Technical Report (TR-OU-INF-2019)

A. Bijlsma[*1], N. Doorn[†2], H. Passier[‡1], H. Pootjes[§1] and S. Stuurman[¶1]

[1] *Open Universiteit, Faculty of Science, Department of Computer Science, Postbus 2960, 6401 DL Heerlen, The Netherlands*

[2] *NHL Stenden University of Applied Sciences, Academy ICT & Creative Technologies, Postbus 2080, 7801 CB Emmen, The Netherlands*

May 27, 2020

[*] lex.bijlsma@ou.nl

[†] niels.doorn@nhlstenden.com

[‡] harrie.passier@ou.nl

[§] harold.pootjes@ou.nl

[¶] sylvia.stuurman@ou.nl

1 Introduction

Introduction. Professional software developers spend a considerable amount of their time on testing activities. In particular, the introduction of agile methods has increased the importance of testing throughout the development process. Yet, many recent graduates lack sufficient testing skills [30]. For example, Edwards found a frightfully low percentage of bugs detected by students of approximately 15% and observed a high application of only happy-path testing [12].

In most university curricula, scant attention is paid to testing. For example, it was found [10] that in only 2 out of 20 Dutch universities testing was fully integrated within the curriculum. In some curricula, testing is not a topic at all. In most curricula, testing education is limited to an introduction of JUnit as Java's popular test framework. How to compose test cases is often a topic that has not been giving enough attention to.

Thus there is a need to pay more attention to testing and to improve the effectiveness of education in the field of testing. Because testing is a complex learning task, according to Van Merriënboer en Kirschner [32], test education should not merely concentrate on factual knowledge, but also provide procedural guidance scaffolding students' activities.

There are strong indications that testing instruction influences the quality of students' programs positively. Some educators state that testing as activity improves software comprehension [11] and motivate that by the introduction and integration of testing in programming education. In fact, it has been observed that knowledge about testing in itself tends to improve the quality of students' programs [22]. This effect was even observed when test cases were provided by the teacher [5]. Moreover, testing is one of the core knowledge areas in both the ACM curriculum guidelines for computer science and software engineering [28, 29].

Research goal. This study is the starting point in which we want to improve our testing education. Ultimately, we aim to improve the quality of the code, and therefore the software, produced by students through better testing education.

Research questions. To this end, the following exploratory research questions should be answered:

- RQ.1. What can be done to improve testing instruction?
- RQ.2. Where should testing be introduced in the curriculum?

A considerable amount of research has already been done on these questions. We will provide literature references in Section 2.

In order to produce better instructional material and procedural guidance, it is important to know how students initially view testing and which misconceptions need to be eliminated. Beizer [3] distinguished five phases in the mastery of testing:

- PHASE 0. Thinking that testing and debugging is the same.
- PHASE 1. Thinking that the purpose of testing is to show that the software works.
- PHASE 2. Thinking that the purpose of testing is to show that the software does not work.
- PHASE 3. Thinking that the purpose of testing is to reduce risks.
- PHASE 4. Thinking that testing is a state of mind, the purpose of which is to develop higher quality software.

we wish to investigate at what level our students are when they have had some exposure to programming but no explicit testing instruction. This leads to the additional research question:

- RQ.3. What ideas do students have about testing before they have had any relevant instruction?

We shall provide an answer to this question via empirical data. In the literature, very little comparable data can be found on this matter. The closest approach we found is the work of Kolikant [20], whose findings we were able to confirm for the most part (but not entirely, see 4.2.2).

This report. In Section 2 the literature concerning our research questions is reviewed. In Section 3 we explain the way our empirical research has taken place. This involved questionnaires, experiments and interviews. The results have been detailed and analyzed in Section 4. We end with a summary of conclusions and proposals for future work.

2 Related work

To be able to integrate testing in the curriculum in such a way that students really get good testing skills, one would like to know, in the first place, how students perform in testing, and what their problems are. In the second place, it is good to review evaluations of ways to integrate testing in the curriculum. We found the following results on both topics.

2.1 How students test

Students seem to use a trial-and-error approach to testing, even when they follow advanced programming courses [11].

How students test has not been examined often, in a systematic way. An exception to this lack of research is the study of Edwards, who analyzed the tests that students had to send in with an assignment (in a course on Data structures) [12]. In this study, grading was based on the branch covering of the tests. Branch covering was good (with a mean coverage of 95.4%). The similarity between the tests was big (90% of the tests were the same). To check which bugs the student's tests could detect, all student-written tests were combined into a single large test suite, along with the instructor-written reference tests for the assignment. This test suite was then run against all student programs. The tests of the students only detected 13.6% of the total number of bugs(!). Almost all students performed so-called 'happy path' testing, implicitly assuming that the input would be 'ideal'.

The fact that software testers in general (not only students but also professionals) tend to rely on 'happy path testing' has been confirmed a long time ago [23]. Instead of 'happy path testing', Leventhal uses the term 'positive bias'. He found strong evidence of positive test bias regardless of condition. The only 'antidote' he could find, consists of thorough and complete program specifications.

The tendency of students only to test the 'happy path' is in accordance with the finding that students have 'alternative standards' for correctness [20]. Students soften the requirement that a function should show correct behavior for *all* input, to the notion that the behavior should be correct for most input, for input that seems 'logical'.

According to practitioners, many recently graduated students lack sufficient testing skills [30]. This study uses a survey and interviews and concludes that practitioners see a skill gap between university graduates and industry expectations and that graduates often do not seem to see the value of testing. Practitioners observe that graduated students often follow a trial-and-error strategy: build something and 'see if it works'.

Explicitly teaching testing does have effect: students who had been taught in the subject produced better test cases [16].

Testing, so it seems, requires explicit attention in the curriculum. Without an explicit specification given by the teacher, students seem to assume a specification that only allows 'ideal' input. It is a well-known fact that students

bring their own assumptions to problems. Their assumptions may, for instance, bring them to view several variables as related while there is no relation in the code [18]. An explicit specification (for beginning students in text) might be helpful to eliminate at least some of these assumptions that students bring to the problem. One should also teach students how to read a specification (for instance, that one may not make any assumptions about what is not mentioned in the specification).

2.2 Where to teach testing in the curriculum?

When one introduces testing early in the curriculum, one faces several challenges. One has to decide where and how testing is introduced, one should find out how to stimulate that students see the value of testing, one should decide the procedure to follow (for instance, use test-driven development or start by stating the specifications), one should find a way to give useful and timely feedback, one should choose the tools for testing, and one should find ways to stimulate that students become better testers [31].

In a study investigating using XP practices to teach object-oriented programming, Keefe et al. found (using observations, student's results, a survey and interviews) that TDD (part of XP) proved to be the most difficult part for students [19]. In particular, JUnit was found to be a stumbling block for weaker students, while stronger students did not see the point of creating JUnit test cases (they were convinced that their code was infallible). One conclusion of the study is that one should first introduce testing, and teach how to write tests, before giving JUnit as a tool.

Writing tests is an activity that the students do not find unpleasant per se [13, 21], but writing tests for their own assigned programs is seen as an unnecessary burden. Scatalon and Garcia [31] point out that in general programs from introductory courses are too simple to justify the additional effort involved in testing; besides, these introductory courses are already very full [1]. Combining these with testing education would necessitate a full redesign [24] and is not just a matter of adding some fragments to an existing course. A different approach would be to awaken students' interest in testing by initially providing them with unit tests rather than having them write these themselves [33].

The importance of introducing one concept at a time (not mixing introducing a tool with introducing testing) was reproduced by Mishra et al. [26]. In their study, strong programming skills appeared to have a positive impact on the success in structural (white-box) and automated testing, but not in functional (black-box) testing – but, indirectly, experience of white-box testing tended to improve the quality of black-box testing too [8]. Indeed, introducing TDD in a programming course seems to have more benefits in advanced courses than in courses for beginning students, as can be seen in the survey that Desai et al. performed [9]. Especially for beginning students, it is important to provide feedback about test coverage, number of unit test passed and so on: if not, they would not write any tests at all. Moreover, it was observed that the benefits in terms of program quality are not dependent on any specific testing strategy

[14].

An argument against introducing TDD early, in courses for beginners, is that expecting beginning programmers to write tests before, or during the time that they learn to code, interferes with the sequence that Bloom's taxonomy suggests [6]. Specifications, they argue, should be *given*, at first. In TDD, tests serve as specifications, and therefore, one forces a student to formulate the specification of a function.

On the other hand, introducing testing early has the benefit that it would lead students to a better development process and attitude [2, 17]. Edwards argues that the trial and error strategy that students use when writing code, can be improved by asking them, from the very beginning, to test their code: assignments consist of writing code and providing the tests for that code. [11]. He discerns several roadblocks for adopting testing in assignments:

- Software testing requires experience with programming (novice students may not be ready).
- Teaching testing requires time, both in terms of lecture hours and in time to assess assignments.
- Students need frequent, concrete feedback to be able to learn.
- Students must see the added value of testing.

One (maybe obvious) conclusion, is that one should teach one subject at a time. When introducing testing in the curriculum, this is easily overseen, and should *not* be overseen. Another conclusion is that to learn testing, students should receive timely, frequent and concrete feedback. In software testing, specification is an important part.

3 Method

3.1 Aim of the study

This explorative study is a baseline measurement on the perceptions of software testing by first-year students computer science. The participants should have programming knowledge on the level of an introductory course about programming, but should have no prior formal education on the topic of software testing. This investigation intends to study their natural way of testing software units during programming. Our study involves inquiries, exercises and interviews. We partially replicated Kolikant's study [20] to be able to compare our results. In this section we describe our research set-up, methods and the way we analyzed the results.

3.2 General approach

First, we started with an inquiry followed by exercises. After the exercises, a second inquiry was held. For these three parts, a duration of 45 minutes was planned. Finally, a subset of the participants was interviewed. These interviews had a duration of about twenty minutes.

3.2.1 Pre-exercises inquiry

The aim of the first inquiry, which can be found in appendix A, is to test the ideas and beliefs the participants have on software testing without having them exposed to the second part, the exercises. This inquiry contains:

- One multiple choice question about at what point in time during development tests can be best formulated, followed by an open-ended question to motivate the given answer;
- Six statements about testing with five point Likert scale answer options indicating the agreement of the participant with these statements.

3.2.2 The exercises

After the pre-exercises inquiry, the participants were presented with four different exercises of which three consist of a functional description and a code listing, and one consists of only a functional description. The exercises can be found in appendix B. For each exercise, the participant is asked:

- to give the test cases needed to be able to decide whether the method is correct or not,
- to determine the correctness of the provided implementation, and, in case of incorrectness, to provide a test case to prove this claim.

The exercises focus mainly on basic types as integers and strings, composite types as arrays, and control flow mechanisms such as if-then-else constructs and while loops.

3.2.3 Post-exercises inquiry

The second inquiry, which can be found in appendix C, is held directly after the exercises. The inquiry asked for:

- the perceived complexity of the exercises,
- the process followed to complete the exercises,
- the supposed correctness of their answers.

During this inquiry, participants were also asked about the general ratio of time spent on programming and testing in daily practice. Finally, the participants are asked if they took boundary values into account during the exercises. All questions of this inquiry have five point Likert scale answer options.

3.2.4 Interviews

A subset of participants is interviewed using a semi-structured interview to obtain more in-depth knowledge of the way the exercises have been done. Furthermore, the interviewers are able to verify the given answers to the exercises and the students' ideas about testing in more depth. All interviews were conducted by two interviewers, one mainly the panel chair and mainly making notes. All interviews were audio-recorded and written down verbatim. The interview guide followed can be found in appendix D.

3.2.5 Participants feedback on the investigation

During both inquiries and the exercises, participants are encouraged to provide remarks and feedback for the researchers.

3.3 Focus of the investigation

In this research, we investigated the ideas the participants have about testing prior to formal education on this topic. We aimed to study both their testing methods and the way they understand the concepts related to testing. We replicated part of the study conducted by Kolikant concerning testing [20]. By conducting this research, we also gained insights into the misconceptions of the participants.

3.3.1 Testing concepts

The following concepts related to testing were investigated:

- During what programming phase is testing relevant?
  - Pre-exercise inquiry: question if testing should be done before, during or after programming.
- Which stakeholder should conduct testing?

  - Pre-exercise inquiry: statement if testing should be done by end-users.

- In what depth and width should software be tested?
  - Pre-exercise inquiry: statement to indicate if test cases should be representative of the expected use of the program.
  - Exercises: the students' created test cases result in the width and depth in which they cover the program's specification and/or implementation.
  - Post-exercise inquiry: statement to indicate if all possible inputs are covered by the test cases.
  - Interview: for each exercise, questions about the motivation of the created test cases and the approach the participant followed in designing them.

- What time ratio should be spent on testing?
  - Post-exercise inquiry: statement about the ratio of time to be spent on testing versus programming (coding) in daily practice.

- How much time was spent on creating the test cases?
  - Exercises: for each exercise, a start and end time is noted.

- Completeness of testing, i.e. when does one have enough test cases?
  - Exercises: the students' created test cases show the completeness.
  - Post-exercise inquiry: statement to indicate the completeness of the created test cases.
  - Post-exercise inquiry: statement to indicate if all possible inputs are covered with the test cases.
  - Post-exercise inquiry: statement about the chances that there might be input values that could make the program behave unexpectedly.
  - Interview: for each exercise questions about the completeness of the created test cases.

- Does creating test cases help with understanding code?
  - Post-exercise inquiry: statements for each exercise to indicate to what extent the exercise was understood.
  - Post-exercise inquiry: statement to indicate to what extent creating test cases helped with understanding the code.

- Does one use boundary values for test cases?
  - Exercises: the created test cases show if boundary values were used.
  - Post-exercise inquiry: statement to indicate to what extent one has used boundary values to create test cases.

3.3.2 Replication of Kolikant

The statements regarding testing (and not about the understanding of correctness) of Kolikant were replicated in both the pre- and post-exercise inquiries. In the pre-exercise inquiry statements 4, 5, 6 are from Kolikant, and in the post-exercise inquiry statements 7 and 8 are from Kolikant. During the interviews, these statements were occasionally checked.

3.4 Test run

Before the actual study, a test run was held to discover vagueness, incompleteness and feasibility issues. Eight students were involved, all second year bachelor computer science students. Seven of these students had already attended the course "Software Testing". The students were able to give feedback on all parts of this investigation, by writing as well as oral.

The test run was very useful. Some questions of the inquiries were not clearly phrased. The text of some exercises was unclear in comparison with the examples of the input-output given. Students had misunderstandings about the black box exercise: the body of the method was deliberately omitted and not forgotten. Last, but not least, some practical issues as problems with audio equipment came to light.

Based on these findings, the inquiries, exercises and interview guide were improved.

3.5 Analysis

All analyses were performed by four researchers, all involved in software engineering education.

Pre- and post-exercises inquiries. Both inquiries were subjected to quantitative analysis. The students' scores were counted and aggregated by means of Excel spreadsheets. After that, simple statistical analyses were performed, based on mean and variance values.

The answers on the open question from the pre-exercise inquiry were collected and analyzed quantitatively. This question asked for a motivation for the answer given to the question about the best phase to design test cases (before/-during/after coding). We validated the answers in terms of relevance, i.e. yes (relevant) or no (not relevant). If an answer was relevant, we attributed it using certain general characteristics we have found in the answers. Examples of these characteristics are:

- Before, during or afterwards testing.
- Primitive integration- and unit tests: does the participant test the whole by testing the individual components.
- Iterative approach: write some code, immediately write tests.

- Trial and error approach.
- Focus on code.
- Focus on functionality.
- Testing as a means to check whether the code is robust.

If an answer was not relevant we did not attribute it. The analysis was performed by two researchers and reviewed by the other two researchers.

Exercises. From the exercises, all answers were collected in an Excel spreadsheet. The answers were then analyzed separately on completeness of the test cases, test approach followed, mistakes made, misconceptions occurred, and time spent. After that, the findings were aggregated by defining a classification, defined by means of a brown paper session. The results of these findings and classification were discussed until consensus of all decisions was reached.

Interviews. The participants were interviewed by two teams of two researchers: one researcher mainly as interviewer, the other mainly making notes. All interviews were audio-recorded and these recordings were transcribed verbatim. Then, all transcripts and notes were read in their entirety by all researchers. Then, we analyzed the interviews with respect to the completeness of the test cases and the approaches followed. The results of these interviews were analyzed qualitatively and aggregated by defining a classification defined through a brown paper session. For each class, examples were determined. The results of these findings and classification were discussed until we obtained consensus of all decisions.

Meta-analysis. Finally, to determine the main findings of this research, we performed a meta analysis. By means of a brown paper session, the most important classes of our findings were determined and provided with clear examples. Again, the results of this analysis and classification were discussed until we obtained consensus of all decisions.

4 Results

4.1 Participants

Thirty-one students were involved, all first-year computer science students at a university of applied science. All students have basic knowledge of:

- HTML, imperative programming using PHP (period 1).
- Databases and SQL, with some attention to exception handling (period 2).
- Introduction to OO programming with Java using BlueJ (period 3).

In all these courses, testing was not a topic. For the inquiries and the exercises, 45 minutes were available. Eleven students were interviewed during an interview of 20 minutes each.

4.2 Pre-exercises survey

In this subsection, we present the results of the survey conducted before the students did any of the (paper-based) exercises. The number of students that filled in the survey is 31.

4.2.1 When to test

Question: *The best time to construct test cases is (a) after, (b) before or (c) during programming? Motivate your answer. (Multiple answers are allowed.)*

Answers: The combinations figuring in the answer were

combinations	frequency ($N = 31$)
a	2
b	2
c	12
ab	0
ac	9
bc	1
abc	5

If we just look at the number of times an alternative was mentioned, possibly in combination with others, we get the following:

alternative	frequency ($N = 31$)
a	16
b	8
c	27

Conclusion: There is a clear preference for testing during programming, which is consistent with the tendency to base test cases on code inspection apparent from the interviews: see Subsection 4.5. The low score for testing before programming is understandable for the same reason. We also suspect, on the basis of motivations added to the answers to this question, that some students do not clearly distinguish between constructing test cases and running tests.

4.2.2 Claims about testing

We presented the students with six statements about testing, several of which were taken from Kolikant [20]. The students could indicate the level of agreement on a five-point Likert scale, where 1 denotes complete disagreement and 5 complete agreement. For these statements, we have $N = 29$.

Absence of errors Claim: *Testing can make it plausible that your program does not contain errors.*

Answers: Average 3.41, standard deviation 1.02. Many students place a lot of trust in the power of testing.

Who tests? Claim: *It is best if end users perform the tests.*

Answers: Average 3.66, standard deviation 1.20. This statement is widely agreed with, which is unexpected in view of the preference expressed above for constructing test cases during development. Possibly the latter preference is based on a confusion of testing with compiling, running or debugging.

Which test cases? Claim: *The most important consideration when selecting test cases is to ensure that they are representative of the expected use of the program.*

Answers: Average 3.59, standard deviation 1.05. This shows the tendency to 'happy path testing'.

Confidence Claim: *For a program I have written myself, I know it works well if I have run it several times and obtained correct output.*

Answer: Average 2.66, standard deviation 0.98. This is similar to claim A.1 from Kolikant's paper [20]. In that paper, 50% of respondents agreed with the statement, both at high school and college level. Our respondents seem to possess a somewhat more sophisticated attitude (only 24% agreed), although this could be induced by the context of the preceding questions. However, in the next questions the difference with Kolikant's findings grows markedly more pronounced.

Reasonable output Claim: *In testing a program for a complicated calculation, I am satisfied if the output looks reasonable. It is not necessary to redo the calculation by hand.*

Answer: Average 1.69, standard deviation 0.70. This is similar to claim A.2 from Kolikant [20]. There 33% of high school students and 69% of college students agreed. Of our respondents, only one could be said to agree (answer 4). For this discrepancy, we have not found a satisfactory explanation. The same goes for the next claim. A reasonable conjecture at this point would be that the students have been influenced by tasks they did earlier, where perhaps redoing the calculation by hand was infeasible. This conjecture is reinforced by our own observation that the answers given for one particular exercise strongly referred to a similar exercise these students had done before in another context. It is also consistent with the difference Kolikant found between high school and college students.

No testing Claim: *Sometimes I am sure that a program I have written is completely correct. In such a case, if the program compiles, it is not necessary to run or test the program.*

Answer: Average 1.55, standard deviation 0.90. This is similar to claim A.3 from Kolikant [20]. There 42% of high school students and 31% of college

students agreed. Of our respondents, only two (or 7%) agreed (both gave the answer 4).

4.3 Exercises analysis

Students were presented with four exercises. In each exercise, they were asked to write down suitable test cases for these four exercises. Three of the exercises were 'black box' as well as 'white box' exercises were both a functional specification and Java code was provided. One exercise was 'black box' with only the functional specification. The exercises can be found in Appendix B.

For each of the white box exercises, students were asked if they considered the code to be correct. If not, they were asked to present a test case that would fail due to the incorrect code.

All exercises were single methods that had input and output in the form of arrays of integers or a single integer. The exercises contained programming constructs and syntax that should be familiar to the students and were part of the previous Java courses they followed. The exercises presented to the students are as follows:

Exercise 1: The longest period of frost This method provides the length of the longest period of frost from a series of temperatures. The input is an array of integers representing the temperatures of a sequence of days, the output is an integer representing the length of the longest number of consecutive days the temperature was below zero degrees Celsius. The body of the method is not correct, i.e. variable `currentPeriod` is initialized to -1 instead to 0.
The code is as follows:

```
/**
 * Returns the longest uninterrupted period of temperatures below 0
 */
public int longestBelowZero(int[ ] temperatures) {
    int longestPeriod = 0;
    int actualPeriod = −1;
    for (int i=0; i < temperatures.length; i++) {
        if (temperatures[i] < 0) {
            huidigePeriode++;
        } else {
            if (actualPeriod >= longestPeriod) {
                longestPeriod = actualPeriod;
            }
            actualPeriod = 0;
        }
    }
    return longestPeriod;
}
```

Listing 1: Frost exercise (frost)

Exercise 2: The lowest index of the lowest value This method has an array with integers as an argument and it's function is to determine the lowest index of the lowest number in the array. The provided code is incorrect, the index in the for loop that iterates over the values is initialized on `1` instead of `0`. This is a classic one-off error.
The code is as follows:

```
/**
 * Returns the lowest index of the lowest value
 */
public int findTheLowestIndexOfTheLowestValue(int[ ] numbers) {
    int index = 1;
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] < numbers[index]) {
            index = i;
        }
    }
    return index;
}
```

Listing 2: Lowest index of the Lowest value exercise (min-min)

Exercise 3: Changing coins This method has an integer as an argument which represents an amount of money in Euro cents. The method returns the shortest possible set of coins that represents that amount of money.
The code is as follows:

```
/**
* Returns the least amount of coins possible to represent the input argument.
* Possible coins:
* 1, 2, 5, 10, 20 and 50 cent
* and
* 1 and 2 euro (100 and 200 cent)
*/
public ArrayList<Integer> exchange(int amount) {
    ArrayList<Integer> result = new ArrayList<>();
    int[ ] coins = {200, 100, 50, 20, 10, 5, 2, 1};
    for (int coin : coins) {
        for (int i=0; i < amount / coin; i++) {
            result.add(coin);
        }
        amount = amount − (amount / coin) * coin;
    }
    return result;
}
```

Listing 3: Exchange exercise (coins)

Exercise 4: Palindrome The provided method has a String argument and returns a Boolean value depending on whether the value is a palindrome or not. This is the black box exercise, so no Java code was given, only the signature of the method.
The code is as follows:

```
/**
 * Input A string
 * Output true if the string is a palindrome
 * otherwise false
 */
public boolean isPalindrome(String word) {
    // no body is provided
}
```

Listing 4: Palindrome checker exercise (palindrome)

These four exercises are diverse in nature but all have an algorithmic nature. Based on the functional specification and on the code, students should be able to understand the algorithm and come up with a test strategy. The exercises differ in the concepts that are used. The frost exercise contains a `for` loop that iterates over the input array. It also has branching a conditional statement (`if else`) with another conditional statement in the `else` branch. The second exercise uses the same array with two indexes in the conditional statement. This can be easily overlooked. The coin exchange exercise uses a nested loop construct. This is often considered to be a complex concept for novice programmers [7, 15]. There is also a mathematical statement with a subtraction, a multiplication and a division. The palindrome exercise, handles string input and uses a Boolean return value.

Categories of observations Analyzing the students' answers, we have found four main categories of observations:

1. the test approaches
2. the completeness of the test cases
3. the misconceptions
4. programming knowledge

For each category, we give some examples where the exercise is mentioned between brackets, i.e. frost, coins, min-min, or palindrome.

4.3.1 Test approaches applied

Happy path testing The most applied test approach by the students is happy path testing. Although happy path testing is a valuable test approach, it is insufficient as this approach is applied only. We determined 33 test sets consisting of happy path test cases only: frost 10, coins 3, min-min: 6, and palindrome: 14. Two examples, the first is from the frost exercise:

'One test case with at least one temperature below zero.'

An example from the min-min exercise:

'[0,1,2,0,2,-1]'

Structural approaches Some students applied techniques that could be interpreted as boundary value testing. We determined nine test cases that could be considered as boundary testing.

One student differentiated on the frost exercise between a test case with one temperature and a test case with several temperatures:

'One test with a known outcome. Then, one array with one item. And one array with temperatures below zero only.'

Another student defined (frost) an empty array and a number of arrays with length greater than zero:

'[], [-1,-1,-1,4,3,-2,-2], [-1,0,1,2,3,], [-1,-1,2,2].'

One student applied a more or less systematic test approach. The student described four test cases (frost):

'An array without temperatures below zero, an array with one period of frost, an array with multiple periods of frost, and an array with a period of frost at the beginning.'

The last test case shows the bug in the code. Nevertheless, the students did not define test cases with array's of length zero and one.

Test cases based on code inspection - bug find We could identify four situations were a student wrote only one test case based on the bug found in the code after code inspection. One example found as part of the frost exercise:

'[-1,-1,0,1]'

The method output is 1, where the longest period of frost is 2, due to the wrong initialization of variable `currentPeriod`.
Another example of a test case based on code inspection is (min-min):

'[..] after that I would use a test case where the lowest value is on the first index, this will probably fail because of the for loop which starts with i=1. Personally, I would probably never test this because I would have noticed this while programming.'

This test case demonstrates the bug present in the code. The last part of the quote underlines the approach of code inspection which looks like an alternative for testing.

The following example points directly to the bug in the code, where the student mentioned this test case only:

'[2,2,2,2,2]'

Remark. A reason why students based their test cases often on code inspection can be stimulated by the question we asked them of considering the correctness of the code, and if not correct, to present a test case that would fail due to this incorrectness.

Miscellaneous One student gave an answer we do not understand (frost), it might be an approach to debugging instead of testing:

'You have to see the array to figure out if it is correct.'

One student was unable to provide a concrete test case. This student gave the following description (frost):

'An input value of which you know the output value'

which is a basically a very high level description of testing in general.

4.3.2 Completeness of test cases

A complete test set should discern several aspects, as for example structural as well as domain specific aspects, and specification as well as implementation based test cases. Specification based test cases are only possible, of course, if an implementation is presented [4].

In case of the frost exercise, examples of structural aspects are an empty array, an array with one element and an array met several elements. Examples of domain-specific aspects are no frost periods at all and frost periods of different lengths spread out over the array in several ways. An example of a specification aspect is what to do in case of an anomaly. If the specification is present, one can think of applying various coverage criteria as well as testing overflow situations in cases specific types are used for variables.

We observed that almost all the test sets defined by the students are far from complete. For example, for the frost exercise, a minimal of three test cases are needed to have path coverage. Only one student provided enough test cases to reach path coverage, consisting of the following three test cases:

'[0,-1,1,-1,0,-1],[-1,0,-1,-1],[1,1,0,1]'

Most students defined one up to four test cases which is not enough to test the given methods sufficiently. For example, for the first exercise, the frequency is: one test case: eleven times, two test cases: four times, three test cases: four times, and four test cases: two times. Furthermore, as mentioned before, most of these test cases test the happy path scenario only. For example, one student defined just one test case (frost):

'One test case with at least one temperature below zero.'

An example of an answer limited to two test cases (frost):

'One test case with a negative number and one test case with a positive number.'

An example of an incomplete test case for the coin exercise:

'Test cases with multiples of coins.'

The students who applied a more systematic testing approach, as mentioned before, had a slightly more complete test set.

One student mentioned that the body of the change method (coins) is incorrect, but was not able to define a test case showing the bug.

4.3.3 Misconceptions

Exhaustive testing One student presented a test case and then proposed an exhaustive testing scenario (frost):

'[-1,-1,0,0,1,1] and I shall look to all possible inputs and see whether the program reacts as is expected with several days of frost.'

Test cases without expected result Some students specified an array with random numbers as a test case. for example

'Random numbers in an array.'

The problem with this approach is that the result of such a test case is unknown beforehand and therefore it is impossible to determine its correctness.
Another example (min-min) also defining a random array as test case is:

'a) One array with equal numbers, b) one array starting with the lowest number, c) one array with all random numbers, and d) one array with numbers you know the result of.'

Type testing One student defined a test case with an array containing a character, where an array with integers is expected (min-min):

'One array with two numbers, one array with a lowest number, and an array with a character.'

The language used is Java, a strongly typed language.

Dividing by zero Two students remarked that dividing by zero is forbidden and as a consequence dividing zero by something else is forbidden as well.[REF??]

Implementation is required As part of the Palindrome exercise, one student wrote one test case ('lol', which is a palindrome), but mentioned that it is impossible to check the case because the implementation is missing.

4.3.4 Programming knowledge

Although these students should have the required knowledge about Java, it seems that some students struggle with a given code. For example, one student wrote as a test case for the coin exercise:

> *'49,7,9,127,61 I think that something is missing with 'int coin', because an int can not be an array.'*

Here, this student, probably, has not enough knowledge of Java types.

Some students are not focused on input-output testing, but on print-based testing to check whether certain statements are successfully executed and in what order. For example, one student wrote as a test case (min-min):

> *'I define a method that prints the array to see the array is successful created.'*

Miscellaneous One student had no idea how to solve this exercise (frost) and stated:

> *'I have no idea.'*

One student did not understand the palindrome exercise, judging by the answer:

> *'droom, paling, moordnilap, true, false, 12345, palindr00m'*

.

4.3.5 Time spent on the exercises

The students were asked to measure the time in minutes it took them to complete the exercise. We got the following averages per exercise:

Exercise	Average time spent	standard deviation
1 ($N = 31$)	6:42	3:07
2 ($N = 30$)	4:30	1:46
3 ($N = 31$)	4:54	1:59
4 ($N = 31$)	2:17	1:16

The first exercise took the students the largest amount of time. This could be caused by the time needed to understand how the exercises worked. During the interviews, students did not consider the first exercise to be the most difficult one, the third exercise was considered to be the most difficult by many students. The time spent on that exercise reflects that difficulty. The last exercise, the black box exercise, took considerably less time than the exercises with the code provided. This supports our findings that students mainly use the code to think of test cases.

Overall, the short time spent by students to solve these exercises strikes us. This finding matches with the findings of happy path testing, test cases based on code inspection, and when a bug is found, a test case for that bug is composed.

4.3.6 Correct or incorrect

For each white box exercise, students were asked if they think the code was correct and if not, to come up with a test case to support their claim. Exercise one and two both contained one logical error and exercise three was correct. None of the exercises contained syntax errors. The following table shows the results:

Exercise	Correct	Incorrect	Valid test case
1 ($N = 22$)	11	11	7
2 ($N = 25$)	4	21	19
3 ($N = 23$)	10	13	n/a

The first exercise shows an equal amount of students who think the code is correct and incorrect. Seven students were able to provide a valid test case to support their claim. On the second exercise, students scored a lot better. Most students noticed the bug and were able to provide a test case to support their claim. With the third exercise, most students wrongly think the code is incorrect. This supports the indication that most students found this exercise the hardest.

4.4 Post-exercises survey

The survey has been filled in by 31 students ($N = 31$).

The following statements were submitted to the students after they had been asked to come up with suitable written test cases for a set of four programming problems. These exercises were presented on paper and the students did not use a computer when answering them.

Understanding Claim: *Having to think of test cases has increased my understanding of the program code.*

Answer: Average 3.42, standard deviation 1.13. We did not verify whether any deeper understanding was actually reached.

Test coverage Claim: *My test cases were sufficient to test the program.*

Answer: Average 3.10, standard deviation 0.83. In fact the test cases were clearly insufficient, which the students only realized when discussing them in the interview phase. It seems many students interpreted the exercise as 'find the coding error in this program' and stopped when they had found one.

Remark. As we have noticed in Section 4.3.1 this can be stimulated by the question we asked them of considering the correctness of the code, and if not correct, to present a test case that would fail due to this incorrectness. However, this is not mentioned in the interviews.

Systematic testing Claim: *I test a program by systematically checking all possible input values.*

Answer: Average 3.67, standard deviation 0.75. This is similar to claim A.4 from Kolikant [20]. There 71% of high school students and 75% of college students agreed. In our population, 79% agreed. This is a remarkable estimate, as the exercise results showed that the students produced a very limited set of test cases that certainly did not cover all possibilities.

Overlooking cases Claim: *There is always the possibility that the program fails for some input value I have not discovered.*

Answer: Average 4.59, standard deviation 0.62. This is similar to claim A.5 from Kolikant [20]. There 54% of high school students and 81% of college students agreed. In our population no less than 93% agreed. This shows that the optimism exhibited in the previous section should not be taken too literally. Kolikant [20] concludes from these numbers that students tend to describe their non-systematic methods as systematic. Our results strongly confirm this conclusion.

Time use Claim: *The ratio of times spent on programming and testing should be (1) 100/0, (2) 75/25, (3) 50/50, (4) 25/75, (5) 0/100.*

Answer: Average 2.64, standard deviation 0.75. Of the respondents 48% needed most time for programming, 41% spent the same amount of time for both, and only 10% found that testing took the most time. (One student had the answer 0% programming, 100% testing!)

Boundary values Claim: *In selecting test cases I take boundary values into account.*

Answer: Average 3.53, standard deviation 0.85. However, it was established in the interviews that not all students know the meaning of the term boundary values. In the exercise results, we see boundary values used only occasionally.

4.5 Analysis of interviews

After the pre-exercises survey, the exercises, and the post-exercises survey, we interviewed eleven students for about twenty minutes per student. In the following paragraphs, we will present our findings and will illustrate these using some phrases from the transcribed interview texts.

4.5.1 Test cases are based on code inspection

A very often mentioned approach for composing test cases is code inspection. The functionality of the code is determined based on the code itself instead of the specifications describing what the code should do. An example of this approach is:

> *'First, I've read the description of the exercise, after which I read the code thoroughly to determine its functionality. Otherwise, I am not able to determine the expected outcomes.'*

This student has read the description of the exercise, but has read the code in detail to be able to determine its functionality. Examples where students indicate explicitly that the code is needed to understand its functionality are:

> *'I read the text above the code and looked at the code to determine if I understood what happens in the code.'*

> *'Here, I used to read the code and hope to get more information on how it should work.'*

That students need the code for composing test cases is shown by the following phrases:

> *'I was surprised that there was no code! That means that you have to think about test cases based on the specifications only!'*

> *'There was no code available, so I have to think about how it works. So I imagined how it could be implemented, to see how it should work.'*

4.5.2 When a bug is found, a test case for that bug is composed

Many students mentioned that they are looking for bugs in the code. For each bug found in the code, a test case is composed. Two examples of this approach are:

> *'Interviewer: And suddenly, you saw the error in the code?*
> *Student: Yes, and then I thought, I write* [1,2,3] *and then it is ready, on to the next one.'*

> *'Because I spent a lot of time on it and could not find any bugs. So I stopped searching for wrong code.'*

Furthermore, students mentioned that, besides happy path testing, the test cases are limited to the bugs found in the code. This can explain the low number of test cases we observe in the students' solutions on the exercises (see Section xxx). An example of this shows the following phrase:

> *'Actually, I devise test cases more or less on what I see in the code, as if to say this is erroneous.'*

Other methods of implementation based testing as the use of coverage techniques, were never mentioned during the interviews.

4.5.3 Application of wrong test strategies

Often mentioned test strategies are: a small number of random based test cases, happy path testing, pursue exhaustive testing, restrict test cases to the examples described as part of the exercise, and restrict test cases to bugs found in the code.

An example of random testing is as follows. On the question of the number of test cases is sufficient, the student answered:

> *'Student: For this, it is enough.*
> *Interviewer: You took some numbers, randomly, and looked ...*
> *Student: ... if they are correct. Yes.'*

An often applied approach is happy path testing. An example is:

> *'It was more about figuring out .... how much ..., how often the longest period of frost took place, say .... when the longest period of frost took place. For testing, you need only negative numbers. If there are no negative numbers, there is no longest period of frost. So ...'*

Some students pursue exhaustive testing.

> *'Only integers are allowed. Thus, in that case all possible integers as input until the computer is not able to process them. That should be a physical problem. Yes.'*

Sometimes, students limited the test cases to the example(s) given in the exercise test.

> *'I used exactly the same examples as given in the exercise text.'*

This often leads to happy path testing too.

Finally, as we have mentioned earlier, students often limit test cases to bugs find in the code. An example of this is:

> *'Interviewer: And suddenly, you saw the error in the code?*
> *Student: Yes, and then I thought, I write* [1,2,3] *and then it is ready, on to the next one.'*

4.5.4 Unnecessary or even impossible testing

Some students mentioned they add test cases to check value types although the program was coded in Java, a strong and static typed language, which means the compiler detects type errors directly. Examples are:

> *'Here I can add some characters and look how the program reacts because the program expects integers, but if I put in characters, then the program should chuck them out.'*

'If I have to input a number, then I input a string as for example 'HELLO' *and see what happens.'*

We consider this as a misconception. Another misconception is that some students consider testing as a way of finding syntax errors.

'It is possible that you forget a semicolon, and yet it does not work. In such a case it is good to look at each line of code and to see where it goes wrong. This is a way of testing.'

These misconceptions probably show students do not understand compiler's principles.

4.5.5 Lack of motivation

Students often don't see the necessity of testing code thoroughly:

'Student: No, if I had a computer, then I should apply much longer test cases.
Interviewer: Is there a reason you did not do that?
Student: Yes, too much effort.'

The following example shows the importance of grading:

'I think that how important is the exercise ..., if it is for grading, then I should perform testing more elaborately then in cases of ordinary looking after the code. That is possible, nevertheless, then it works, but in cases of grading, then you should find all errors in the code.'

The following example is related to attitude/engagement:

'I do not find myself good. It was early in the morning. Is possible that I missed some things. The attitude I made the exercise with played a role too. For me, this research is not important, it is not my research.'

4.5.6 Reading someone else's code is difficult

Students often mentioned that reading someone else's code is difficult. Examples are:

'I experienced a lot of problems with the code conventions because I am used the place the brackets in a different way.'

'I did not understand the code really, because is naturally not my own code.'

This could be a reason for the few test cases students wrote. This is in accordance with Kolikant's finding, that students often avoid complexity [20].

4.5.7 Pen and paper versus working on a computer

Some students mentioned explicitly that they prefer working on a computer instead of working with pen and paper. Working on a computer means running the code to see if it 'works'.

> *'It is difficult for me to do it just with pen and paper. It is easier to do it on a computer. Then, you can easily see what happens while running the code, what the code exactly does.'*

Students look at bugs by experimenting with the code, for which a computer is needed. With pen and paper, this approach is not possible. In fact, they debug the code instead of test it. This is an educational issue: students have to learn the differences between debugging and testing and have to learn how to write specification-based tests, probably the best with pen and paper.

5 Conclusions and Future work

5.1 RQ.3. Students' ideas about testing

Our main goal is to improve the quality of the code produced by students through better testing education. In order to improve our test education, we need insight into student's misconceptions and their view on testing before they have had any relevant instruction about this topic. Concerning our third research question, our findings are:

Test cases are based on code inspection. Four of our programming exercises had both an informal specification and code, the last exercise (the palindrome), only had a specification. It was remarkable that, even in the case of the palindrome exercise, students based their tests on code inspection. For the palindrome exercise, they first thought about the code they would write to solve the problem, or thought about an implementation they made earlier for example during another part of the educational program. The notion of basing tests on the specification was not present among the students.

Some students could not write tests at all for some exercises because they did not understand the code of the exercise, which was written by somebody else. The idea that one can write test cases solely based on a specification is totally unknown to the students. So in the case of having problems with reading the code, they did not try to write tests based on the specification.

When a bug is found, a test case for that bug is composed. Students often use testing as a means to find or show bugs in the code. During the interviews, and based on the test cases defined during the exercises, we see that students read the code, find a bug and write a test case for that bug as opposed to apply a structured way to test all relevant scenarios. This can at least partially be attributed to the test setup. During the exercises, students were asked if they think the code was correct and to write a test case that shows the bug if they believe the code was incorrect. This question can be a trigger to specifically search for bugs. It is also apparent that many students moved on to the next exercise after designing a test for the (presumed) bug in the code. They did not take the time to think of other test cases. This strong focus on the given code shows that many of the students do not write test cases as a way to assure the correctness of a program during its complete life cycle, but more of a way to debug the code. This is consistent with the findings of Edwards regarding a trial-and-error approach to software development and testing [12]. It is known from the literature that beginning students do not see a difference between testing and debugging [27].

There is a lack of systematic testing. Both the interviews and the exercise answers show that the students tend to limit themselves to 'happy path testing'. In other words, most students pay no attention to robustness and merely assume

that the input will contain no errors. This finding fits with the inquiry results showing that students are optimistic about the correctness of their code.

This phenomenon is known from the literature [12]; however, we were working with students without any previous training or experience in designing tests, and sensitizing students to the importance of preparing for erroneous input should play a major role in the way testing will be introduced in their education. Apparently, happy path testing is their intuitive way of testing.

In the classification of Michaeli [25], they have a 'level 1' understanding of software quality. In the classification of Beizer [3] they are in phase 1 or in some cases phase 2 of their testing evolution.

A more extreme misconception was found in the case of students who did not think at all about providing test cases, but merely copied the examples that were mentioned in the exercise text for the purpose to illustrate or clarifying the specification. This may be ascribed to misunderstanding the task.

Incomplete test sets. The exercises reveal that almost all the test sets defined by the students are far from complete, mostly only containing happy path test cases. Specification-based requirements (such as robustness), as well as implementation-based requirements (such as coverage ratios) are not satisfied. The above findings explain these incomplete test sets well.

Wrong test strategies. Besides restricting to happy path testing, we also observed other incorrect test strategies. Strategies observed are: random based test cases, test cases restricted only to the examples as part of the exercise, and test cases restricted to bugs found in code. Another remarkable test strategy we observed is exhaustive testing, i.e. trying to feed a function with all possible inputs. This is a known misconception: Complete Testing is Possible[1]. Remarkable is that these students described this approach, but did not try to show their test cases. One student mentioned the possibility of physical problems.

Lack of motivation. Many students showed a lack of motivation for testing. They are optimistic about the correctness of their own code and considered testing merely an additional burden. One reason for this may be that the test tasks presented are experienced as too simple to justify the extra work [31], while code inspection is still feasible. This leads to a paradox in testing education, if the code is small enough to understand, testing is not a necessity. If the code becomes larger, students are unable to comprehend the code and are therefore unable to design tests.

Time spent to test. The time spent by students to read an exercise, to define test cases and to inspect the code is remarkably short. This observation matches the findings of happy path testing, test cases based on code inspection specifying a test case for found bugs, as well as a lack of motivation.

[1] See `https://www.tutorialspoint.com/software_testing/software_testing_myths.htm`

Unnecessary or even impossible testing. Although the language used is Java, which is a strongly typed language, some students proposed type testing in their answers. For example, an array with characters as input where an array with integers is expected. Possibly, students tested the program on robustness, i.e. how it reacts to erroneous inputs. Also, some students used testing as a way to find syntax errors. Because type checking and syntax checking are performed by the compiler, we consider these as misconceptions, i.e. unnecessary testing. As far as we know, this type of misconception is unknown in the literature and can be researched further.

Regarding Kolikant's findings. Regarding Kolikant's study [20], our population of students reveals more mistrust concerning the correctness of a program based on reasonable output of that program: 24% of our population versus 50% of the population of Kolikant consider reasonable output to be a sufficient indicator of correctness. The difference increases in case of complicated calculations: our population 3% versus Kolikant 33% in case of high school students and 69% in case of college students.

A similar observation was done involving the no-testing approach in case a programmer is sure that his/her program is correct. In our study, only 7% of the population agreed that testing is not necessary if the code compiles, where in the Kolikant study 31% agreed with that statement. These findings follow from the pre-exercises surveys.

Almost similar to Kolikant, we observe that 79% of the students think that they test systematically. The exercises and interviews show that they produced a very limited set of test cases that certainly did not cover all possibilities. We also conclude that students tend to describe their non-systematic methods as systematic.

5.2 RQ.1. What can be done to improve testing instruction?

Instead of working on small exercises only, students should work on more complex exercises that include requirements regarding quality aspects such as correctness and robustness. More complex exercises together with these requirements stimulate students to apply testing to check their implementations on correctness and robustness.

The natural way of testing by students proved to be happy path testing and test cases based on code inspection. It is important to emphasize that happy path testing is a meaningful test approach, namely for testing a piece of software in cases were expected input is provided, also named as the 'main success scenario'. Students should continue to apply this type of testing. However, they have to learn how to apply happy path testing in a sufficient way, for example, test cases providing all relevant expected classes of input parameters, and, test cases on input values beyond the boundaries of the expected values. Something similar concerns test cases based on code inspection. Inspection of code provides

the basis for implementation-based testing. Again, students have to learn that testing is different from bug-finding and tests should satisfy certain coverage criteria.

Attention should be paid on the misconceptions found, such as wrong test strategies (for example only happy path testing), unnecessary (for example type testing) and impossible testing (for example exhaustive testing and testing without an expected result).

Furthermore, more attention should be paid to the use of specifications. Specifications are needed both to implement functionality and to develop test cases. This ensures that the test cases test whether the implementation satisfies the specification.

Any treatment of testing in the curriculum must state explicitly what can be achieved by testing. Students are overly optimistic about the correctness of their own code as well as about the possibility of exhaustive testing.

Inexperienced students tend to confuse testing with debugging, exclusively triggered by code fragments seen as dangerous. Testing education should make it clear that in this way errors caused by faulty reasoning will remain undetected.

5.3 R.Q.2. At what point should testing be introduced in the curriculum?

From the literature, we found that before effective specification-based testing can occur, it must be clear what behavior is expected from the program, particularly in case of unusual or erroneous input. Therefore testing should be taught simultaneously with writing or at least reading (informal) specifications. As an alternative, the specifications can be presented in the form of a unit test drawn up by the teacher. These specifications can help students in paying attention to specification-based testing besides testing based on code-inspection and bug finding.

5.4 Future work

We see two main directions for future research. First, one can investigate in more depth the nature and the origin of misconceptions students have. For example, the misconception 'unnecessary or even impossible testing' may be caused by a lack of knowledge or by a wrong understanding of programming language concepts. Based on these new insights, programming and testing education can be improved.

Second, the results so far indicate that the use of specifications might be an important part of supporting students developing correct and reliable software. It might help students in not only creating test cases during code inspection and bug finding, but also taking specifications explicitly in consideration. Specifications can help students in creating test cases in a more systematic way. For example, specifications express precisely the data that are expected as input in terms of types and boundaries. We will develop fine-grained procedural guidance for several procedures: for example to read specifications, to draw up

specifications, and to use specifications for implementing the functionality and to create test cases.

References

[1] Joel Adams. Test-driven data structures: Revitalizing cs2. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 143–147. ACM, 2009.

[2] Ellen F. Barbosa, Marco A.G. Silva, Camila K.D. Corte, and José C. Maldonado. Integrated teaching of programming foundations and software testing. In *2008 38th Annual Frontiers in Education Conference*, pages S1H–5. IEEE, 2008.

[3] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, second edition, 1990.

[4] A. Bijlsma, H.J.M. Passier, H.J. Pootjes, and S. Stuurman. Integrated test development: An integrated and incremental approach to write software of high quality. In Vreda Pieterse, George Papadopoulos, Dave Stikkolorum, and Harrie Passier, editors, *Proceedings of the 7th Computer Science Education Research Conference (CSERC)*, pages 9–20. ACM, 10 2018.

[5] Maria A.S. Brito, João L. Rosi, Simone R.S.d̃e Souza, and Rosana T.V. Braga. An experience on applying software testing for teaching introductory programming courses. *CLEI Electronic Journal*, 15(1), 2012.

[6] Duane Buck and David J Stucki. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. *ACM SIGCSE Bulletin*, 32(1):75–79, 2000.

[7] Ibrahim Cetin. Students' understanding of loops and nested loops in computer programming: An apos theory perspective. *Canadian Journal of Science, Mathematics and Technology Education*, 15(2):155–170, 2015.

[8] Peter J. Clarke, Debra L. Davis, Raymond Chang-Lau, and Tariq M. King. Impact of using tools in an undergraduate software testing course supported by wrestt. *ACM Trans. Comput. Educ.*, 17(4):18:1–18:28, August 2017.

[9] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2):97–101, 2008.

[10] N. Doorn. How can more students become 'test-infected': current state of affairs and possible improvements. Master's thesis, Open Universiteit, 2018.

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

[12] Stephen H Edwards and Zalia Shams. Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 171–176. ACM, 2014.

[13] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 688–697. IEEE Computer Society, 2007.

[14] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31(3):226–237, 2005.

[15] David Ginat. On novice loop boundaries and range conceptions. *Computer Science Education*, 14(3):165–181, 2004.

[16] Omar S Gómez, Sira Vegas, and Natalia Juristo. Impact of cs programs on the quality of test cases generation: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 374–383. ACM, 2016.

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

[18] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pages 107–111. ACM, 2010.

[19] Karen Keefe, Judithe Sheard, and Martin Dick. Adopting xp practices for teaching object oriented programming. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 91–100. Australian Computer Society, Inc., 2006.

[20] Yifat Ben-David Kolikant. Students' alternative standards for correctness. In *Proceedings of the first international workshop on Computing education research*, pages 37–43. ACM, 2005.

[21] Daniel E. Krutz, Samuel A. Malachowsky, and Thomas Reichlmayr. Using a real world project in a software testing course. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 49–54. ACM, 2014.

[22] Otávio Augusto Lazzarini Lemos, Fábio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. The impact of software testing education on code reliability: An empirical assessment. *Journal of Systems and Software*, 137:497–511, 2018.

[23] Laura Marie Leventhal, Barbee Eve Teasley, and Diane Schertler Rohlman. Analyses of factors related to positive test bias in software testing. *International Journal of Human-Computer Studies*, 41(5):717–749, 1994.

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

[25] Tilman Michaeli and Ralf Romeike. Addressing teaching practices regarding software quality: Testing and debugging in the classroom. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, pages 105–106. ACM, 2017.

[26] Deepti Mishra, Sofiya Ostrovska, and Tuna Hacaloglu. Exploring and expanding students' success in software testing. *Information Technology & People*, 30(4):927–945, 2017.

[27] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, volume 40, pages 163–167. ACM, 2008.

[28] The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM), IEEE Computer Society. *Curriculum Guidelines for Undergraduate Programs in Computer Science*. ACM, 2013.

[29] The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM), IEEE Computer Society. *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Computing Curricula Series. ACM, 2014.

[30] Raphael Pham, Stephan Kiesling, Leif Singer, and Kurt Schneider. Onboarding inexperienced developers: struggles and perceptions regarding automated testing. *Software Quality Journal*, 25(4):1239–1268, 2017.

[31] Lilian Passos Scatalon, Ellen Francine Barbosa, and Rogerio Eduardo Garcia. Challenges to integrate software testing into introductory programming courses. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2017.

[32] Jeroen J.G. van Merriënboer and Paul A. Kirschner. *Ten Steps to Complex Learning, a systematic appraoch to four-component instructional design*. Taylor & Francis, second edition, 2013.

[33] Jacqueline L Whalley and Anne Philpott. A unit testing approach to building novice programmers' skills and confidence. In *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114*, pages 113–118. Australian Computer Society, Inc., 2011.

A Pre-exercise inquiry

Nummer student: .....

Enquête A

Deze enquête is onderdeel van een onderzoek naar de manier waarop studenten software testen. Er zijn geen goede of foute antwoorden, maar het is wel belangrijk dat je zorgvuldig deze vragen beantwoordt. Deze enquête bestaat uit één vraag en 6 stellingen. Vergeet niet het nummer op te schrijven dat je van de docent hebt gekregen. Alle gegevens zullen vertrouwelijk worden behandeld en voor het onderzoek worden ge-anonimiseerd.

Vragen

Testen kun je het beste opstellen (meerdere antwoorden mogelijk):

a) na het programmeren

b) voor het programmeren

c) gedurende het programmeren

Geef in maximaal drie zinnen een motivatie voor de gekozen antwoorden van de vorige vraag:

.....

.....

.....

.....

.....

.....

1

Stellingen

Geef aan in hoeverre je het eens bent met de volgende stellingen.

		Helemaal mee oneens	mee oneens	neutraal	mee eens	helemaal mee eens
1	Met testen kun je aantonen dat je programma geen fouten bevat.	○	○	○	○	○
2	Het werkt het beste wanneer de eindgebruiker een programma test.	○	○	○	○	○
3	Het belangrijkste voor testgevallen is dat ze representatief zijn voor het te verwachte gebruik.	○	○	○	○	○
4	Ik weet dat een programma dat ik zelf heb geschreven goed werkt, wanneer ik het meerdere keren (bijvoorbeeld drie keer) run en daarbij de juiste uitvoer krijg.	○	○	○	○	○
5	Als ik een programma test dat een complexe berekening uitvoert, ben ik tevreden als de uitvoer er ongeveer zo uit ziet zoals ik had verwacht. Het is dan namelijk niet nodig om het te verwachten resultaat met de hand na te rekenen.	○	○	○	○	○
6	Soms weet ik zeker dat een programma dat ik geschreven heb correct zal werken. Wanneer zo'n programma compileert is het voor mij niet echt meer nodig om het uit te voeren.	○	○	○	○	○

B Exercises

Nummer student: .....

Oefeningen

Onderstaande oefeningen zijn onderdeel van een onderzoek naar de manier waarop studenten software testen. Er zijn geen goede of foute antwoorden, maar het is wel belangrijk dat je zorgvuldig deze oefeningen maakt. Geef s.v.p. aan hoeveel tijd iedere oefening je heeft gekost om te maken. Vergeet ook niet het nummer op te schrijven dat je van de docent hebt gekregen. Alle gegevens zullen vertrouwelijk worden behandeld en voor het onderzoek worden ge-anonimiseerd.

Deze test bestaat uit vier opdrachten.

Alvast bedankt voor je medewerking!

1

Oefening 1 van 4: Langste vorstperiode

Begintijd: .....

De onderstaande methode heeft een lijst van temperaturen van opeenvolgende dagen als invoer. De methode geeft de *lengte* van de langste vorstperiode terug. Er is sprake van vorst als de temperatuur lager is dan 0 graden.

Voorbeeld: de langste vorstperiode bij een reeks van de volgende temperaturen $[1, 1, 0, -1, -2, -3, 1, 0, -1, -1, 1]$ is 3, namelijk: $[-1, -2, -3]$.

```
/**
 * Geeft de langste aaneengesloten periode terug van temperaturen onder 0
 */
public int langstePeriodeTemperatuurOnderNul(int[ ] temperatuurPerDag) {
    int langstePeriode = 0;
    int huidigePeriode = -1;
    for (int i=0; i < temperatuurPerDag.length; i++) {
        if (temperatuurPerDag[i] < 0) {
            huidigePeriode++;
        } else {
            if (huidigePeriode >= langstePeriode) {
                langstePeriode = huidigePeriode;
            }
            huidigePeriode = 0;
        }
    }
    return langstePeriode;
}
```

Geef bij onderstaande opgaven als testgevallen de invoerwaarden aan.

Opgaven:

- Welke testgevallen heb je minimaal nodig om na te gaan of de implementatie correct is?
- Is de implementatie correct? Zo nee, geef een testgeval die dat aantoont.

Eindtijd: .....

Oefening 2 van 4: Kleinste index van de kleinste waarde

Begintijd: .....

Deze methode krijgt een lijst met gehele getallen als invoer, en bepaalt vervolgens de kleinste index van de kleinste waarde van de lijst.

Voorbeeld: bij een invoer van $[2, 2, 2, 1, 1, 2, 1, 2]$ is het gewenste resultaat 3, omdat de kleinste waarde (1) het eerste voor komt op index 3.

```
/**
 * Geeft de kleinste index van de kleinste waarde terug
 */
public int vindKleinsteIndexVanDeKleinsteWaarde(int[ ] getallen) {
    int index = 1;
    for (int i = 1; i < getallen.length; i++) {
        if (getallen[i] < getallen[index]) {
            index = i;
        }
    }
    return index;
}
```

Geef bij onderstaande opgaven als testgevallen de invoerwaarden aan.

Opgaven:

- Welke testgevallen heb je minimaal nodig om na te gaan of de implementatie correct is?
- Is de implementatie correct? Zo nee, geef een testgeval die dat aantoont.

Eindtijd: .....

Oefening 3 van 4: Van bedrag in centen naar munten

Begintijd: .....

De onderstaande methode heeft een bedrag in eurocenten als invoer. De methode bepaalt vervolgens de kortste reeks munten die overeen komt met de waarde van het ingevoerde bedrag.

Bijvoorbeeld: de kortste reeks van munten waarmee je 291 eurocent kunt uitbetalen is {200, 50, 20, 20, 1} (een munt van 2 euro, een van 50 cent, twee van 20 cent et cetera).

```
/**
* Deze methode geeft het kleinste aantal munten terug waarmee
* het invoerbedrag wordt samengesteld.
*
* mogelijke munten:
* 1, 2, 5, 10, 20 en 50 cent
* en
* 1 en 2 euro (100 en 200 cent)
*/
public ArrayList<Integer> wissel(int bedrag) {
    ArrayList<Integer> resultaat = new ArrayList<>();
    int[ ] munten = {200, 100, 50, 20, 10, 5, 2, 1};
    for (int munt : munten) {
        for (int i=0; i < bedrag / munt; i++) {
            resultaat.add(munt);
        }
        bedrag = bedrag - (bedrag / munt) * munt;
    }
    return resultaat;
}
```

Geef bij onderstaande opgaven als testgevallen de invoerwaarden aan.

Opgaven:

- Welke testgevallen heb je minimaal nodig om na te gaan of de implementatie correct is?
- Is de implementatie correct? Zo nee, geef een testgeval die dat aantoont.

Eindtijd: .....

Oefening 4 van 4: Palindroom

Begintijd: .....

Een palindroom is een woord dat omgekeerd hetzelfde is. Bijoorbeeld het woord "lepel", omgedraait is dat ook "leper".

De onderstaande methode krijgt een string als invoer en bepaalt of de invoer een palindroom is. Van deze methode is bewust de body niet gegeven, alleen de 'signatuur', je kunt dus wel zien hoe je de methode aan kunt roepen en wat voor soort waarde je terug krijgt.

Voorbeeld: bij een invoer van "lepel" is het gewenste resultaat true, bij "cola" false.

```
/**
 * Input Een string
 * Output true als de string een palindroom is
 * anders false
 */
public boolean isPalindroom(String woord) {
    // body van deze methode is bij deze afdruk bewust weggelaten omdat het bij deze oefening niet om de c
}
```

Geef bij onderstaande opgaven als testgevallen de invoerwaarden aan.

Opgave:

- Welke testgevallen heb je minimaal nodig om na te gaan of de implementatie correct is?

Eindtijd: .....

Geef ons feedback

Mochten je opmerkingen hebben over deze opgaven, zowel positief als negatief, schrijf je feedback dan bij de uitwerkingen.

C Post-exercise inquiry

Nummer student: .....

Enquête B

Deze enquête is onderdeel van een onderzoek naar de manier waarop studenten software testen. Er zijn geen goede of foute antwoorden, maar het is wel belangrijk dat je zorgvuldig deze vragen beantwoordt. Deze enquête bestaat uit 10 stellingen. Vergeet niet het nummer op te schrijven dat je van de docent hebt gekregen. Alle gegevens zullen vertrouwelijk worden behandeld en voor het onderzoek worden ge-anonimiseerd.

Stellingen

Geef aan in hoeverre je het eens bent met de volgende stellingen.

	Helemaal mee oneens	mee oneens	neutraal	mee eens	helemaal mee eens	niet van toepassing
1 Ik vond oefening één (Geld wisselen) gemakkelijk te begrijpen.	○	○	○	○	○	○
2 Ik vond oefening twee (Langste Vorstperiode) gemakkelijk te begrijpen.	○	○	○	○	○	○
3 Ik vond oefening drie (kleinste index van het kleinste getal) gemakkelijk te begrijpen.	○	○	○	○	○	○
4 Ik vond oefening vier (Palindroom) gemakkelijk te begrijpen.	○	○	○	○	○	○
5 De oefeningen om testgevallen te bedenken hebben geholpen om de code van de oefeningen die ik heb gemaakt beter te begrijpen.	○	○	○	○	○	○

1

	Helemaal mee oneens	mee oneens	neutraal	mee eens	helemaal mee eens
6 Ik heb genoeg testgevallen bedacht om de code te testen.	○	○	○	○	○
7 Ik test een programma door systematisch alle mogelijke invoerwaardes te controleren.	○	○	○	○	○
8 Er bestaat altijd een mogelijkheid dat er een invoerwaarde bestaat waarmee het programma niet goed werkt, die ik niet gevonden heb.	○	○	○	○	○

	100-0	75-25	50-50	25-75	0-100
9 Wat denk je dat een goede verhouding is tussen de tijd die je aan programmeren en de tijd die je aan testen besteedt? Het eerste percentage staat voor de programmeertijd; het tweede percentage staat voor de tijd die je aan testen besteedt.	○	○	○	○	○

	nooit	soms	soms wel/soms niet	meestal	altijd
10 Ik houd met mijn testgevallen rekening met grensgevallen[1].	○	○	○	○	○

D Interview guideline

Interview

Dit interview is onderdeel van een onderzoek naar de manier waarop studenten software testen. Dit interview moet opgenomen worden om later te kunnen analyseren. Iedere student heeft een nummer gekregen van de docent. Dit nummer refereert aan de gemaakte opgaven en de ingevulde vragenlijsten.

Snel of aansluitend na het maken van de opgaven dient dit interview afgenomen te worden.

Algemene vragen

1. Welk nummer heb je van de docent gekregen?
2. Heb je er bezwaar tegen dat dit interview wordt opgenomen?

Lijst van de gemaakte opgaven

De student heeft de volgende opgaven gemaakt:

1. Langste vorstperiode
2. Kleinste index van de kleinste waarde
3. Van bedrag in centen naar munten
4. Palindroom (De blackbox opgave)

Interviewvragen per opdracht

1. Kun je in eigen woorden vertellen waar opgave [*naam van opdracht*] over ging?
2. Hoe moeilijk vond je deze opdracht?
3. Wat was bij deze opgave je aanpak om de testgevallen te bedenken?
   (a) Heb je gekeken naar de specificatie/omschrijving van de opdracht?
   (b) Heb je gekeken naar de code (n.v.t. bij Palindroom)?
   (c) Of heb je naar beide gekeken (n.v.t. bij Palindroom)?
4. Wat is het eerste testgeval dat je bedacht hebt en waarom?
5. Welke uitbreidingen van testgevallen heb je bedacht en waarom?
6. Zijn het voldoende testgevallen of hadden het er meer of minder moeten zijn om goed te kunnen testen? Licht toe.

Afsluiting

1. Heb je voor ons nog opmerkingen?
2. Hartelijk bedankt voor je deelname!