Date of acceptance Grade

Instructor

Experiences from Teaching Automated Testing with CrowdSorcerer

Vilma Kangas

Helsinki June 2, 2020

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Vilma Kangas

Työn nimi — Arbetets titel — Title
Experiences from Teaching Automated Testing with CrowdSorcerer

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	June 2, 2020	40 pages + 2 appendix pages

Tiivistelmä — Referat — Abstract

Software testing is an important process when ensuring a program's quality. However, testing has not traditionally been a very substantial part of computer science education. Some attempts to integrate it into the curriculum has been made but best practices still prove to be an open question.

This thesis discusses multiple attempts of teaching software testing during the years. It also introduces CrowdSorcerer, a system for gathering programming assignments with tests from students. It has been used in introductory programming courses in University of Helsinki.

To study if the students benefit from creating assignments with CrowdSorcerer, we analysed the number of assignments and tests they created and if they correlate with their performance in a testing-related question in the course exam. We also gathered feedback from the students on their experiences from using CrowdSorcerer.

Looking at the results, it seems that more research on how to teach testing would be beneficial. Improving CrowdSorcerer would also be a good idea.

ACM Computing Classification System (CCS):
Human-centered computing → Collaborative content creation
Social and professional topics → Computing education
Information systems → CrowdSourcing

Avainsanat — Nyckelord — Keywords
software testing, computer science education, software development

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Software testing means the practices and techniques to make sure that the software product is what it is meant to be. Traditionally, it has not been a very important or integral part of computer science education. There are many reasons for that: the lack of basic programming skills, the curriculum being already full, and a lack of consensus in the community of computer science educators on how to integrate testing into the courses [1, 2, 3].

However, it is important to develop software that fulfills the requirements and the needs of the customer. Testing is the most common way to ensure that those requirements are met. Therefore teaching software testing to computer science students is important.

There has been a number of attempts to do that efficiently. Software testing has been integrated to the curriculum by, for example, having the students to give their own test cases for the programming exercises they submit. Then, in addition to the programs they give, their tests are also evaluated by checking if they find the defects injected to their code, for example. Gamifying the testing process [4] and using automated testing services such as the Web-CAT Grader [5] are also suggested as ways to teach testing, to mention a few.

This thesis introduces a system called CrowdSorcerer. It is designed to gather programming assignments made by students [6]. CrowdSorcerer can be embedded into materials of programming courses. The students write the assignments, the source code and they also create the test cases. The system then checks that the code compiles and that the tests pass. After that the assignments are peer reviewed by other students. Similar tools to gather student created assignments have been used before. Those tools include, for instance, PeerWise [7], CodeWrite [8] and StudySieve [9]. CrowdSorcerer is mostly related to CodeWrite.

In addition to describing the system, in this thesis we studied whether it is possible to teach testing by letting the students create programming assignments with tests using CrowdSorcerer. The created assignments, including the test cases, were collected and analysed as well as the number of test cases for each assignment. We also studied whether there was a correlation between the use of CrowdSorcerer and a testing-related exam question. Lastly, we gathered feedback on CrowdSorcerer from the students to study how the system could be improved.

A similar study where the correlation between the use of CrowdSorcerer and the performance in a testing-related exam question was conducted in 2019 by Kangas et al. [10]. In the study, metrics such as test count, line coverage and mutation coverage, were computed from the test code generated using the user-provided inputs and outputs. Those metrics were then compared to the points received from the exam, especially from the testing-related question. The relationship between students' previous programming experience and their use of the tool and performance in the exam was also studied.

CrowdSorcerer has also been covered in two other papers: Pirttinen et al. presented the tool in 2018 [6] and the students' peer reviews have also been analyzed by Pirttinen et al. [11].

The analysis in this thesis provides additional insight into the previously mentioned articles, describes CrowdSorcerer in a more detailed manner, and outlines future directions for the system based on feedback from students.

This thesis is organized as follows. First, in Section 2, some background is given by introducing literature on the subject. The system description and research context and approach are described in Sections 3 and 4. The results of the study are reported in Section 5, and they, as well as the limitations of the study, are discussed in Section 6. Finally, the conclusions and future work can be found in Section 7.

2 Background

In this section software quality and software testing are defined and the basic concepts of them are explained. How testing can be done and its role in a software development process are also covered. Finally, some ways to teach software testing in software engineering education are introduced.

2.1 Software quality

The goal for software developers is to make their product as good as possible as it increases its value. The higher the quality of a piece of software, the easier it is to maintain, for example.

But how can quality be measured? It seems like there is no clear consensus on this even though quality is considered important and plenty of money and effort has been invested in order to improve it [12].

The international community of people working on software systems has put together some definitions to explain software quality. For example, The IEEE Standard for Software and System Test Documentation defines quality as "the degree to which a system, component, or process meets specified requirements" [13].

On the other hand, ISO/IEC 25010 defines software quality by partitioning it into two models: *quality in use model*, which consists of effectiveness, satisfaction, freedom from risk and context coverage, and *product quality model*, which comprises functional sustainability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability [14].

Testing, which is covered in the next subsection, is one of the most common ways to ensure software quality. There are also other actions developers can take to assure good quality, including conducting formal technical reviews [15, 16], peer reviews [17, 18], pair programming [19] and putting good programming practices such as refactoring into practice [20].

2.2 Software testing

In this subsection software testing is defined. It's role in a software development process is also described. How to measure the quality of the tests is also studied.

2.2.1 Definition and objectives

The International Software Testing Qualifications Board (ISTQB) defines testing in its glossary as "the process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects" [21].

The ISTQB has also formed a Foundation Level Syllabus that provides basic information of software testing [22]. According to the syllabus, some of the objectives of testing are:

- Evaluation of requirements, users stories, design, code, etc.
- Verifying that the requirements for project the have been fulfilled
- Validating the completeness of the project
- Building confidence of the quality
- Preventing bugs
- Finding defects
- Providing information to stakeholders especially regarding the level of quality of the test object

To answer the question of which skills are required from a software tester, the IEEE Computer Society has put together a Guide to the Software Engineering Body of Knowledge (SWEBOK) [23] that describes the knowledge that is generally accepted in the field of software engineering, i.e. knowledge that would be useful for a competent software engineer to have. It goes through the basic concepts of each of its 15 knowledge areas, one of which is software testing.

The Software Testing Knowledge Area is partitioned into 6 topics that contain subtopics. The topics are the following:

- **Software Testing Fundamentals**, which covers the basics of software testing. Its subtopics are Testing-Related Terminology, Key Issues, and Relationship of Testing to Other Activities.
- **Test Levels**, which consist of two subtopics: The Target of the Test and Objectives of Testing.

- **Test Techniques**, which covers the most common testing techniques: Input Domain-Based Techniques, Code-Based Techniques, Fault-Based Techniques, Usage-Based Techniques, Model-Based Techniques, Techniques Based on the Nature of the Application, and Selecting and Combining Techniques.
- **Test-Related Measures**, the subtopics of which are Evaluation of the Program Under Test and Evaluation of the Tests Performed.
- **Test Process**, which covers how the testing activities can be supported and guided. Its subtopics are Practical Considerations and Test Activities.
- **Software Testing Tools**, which goes through Testing Tool Support and Categories of Tools.

The hierarchy of these topics is presented in Figure 1.

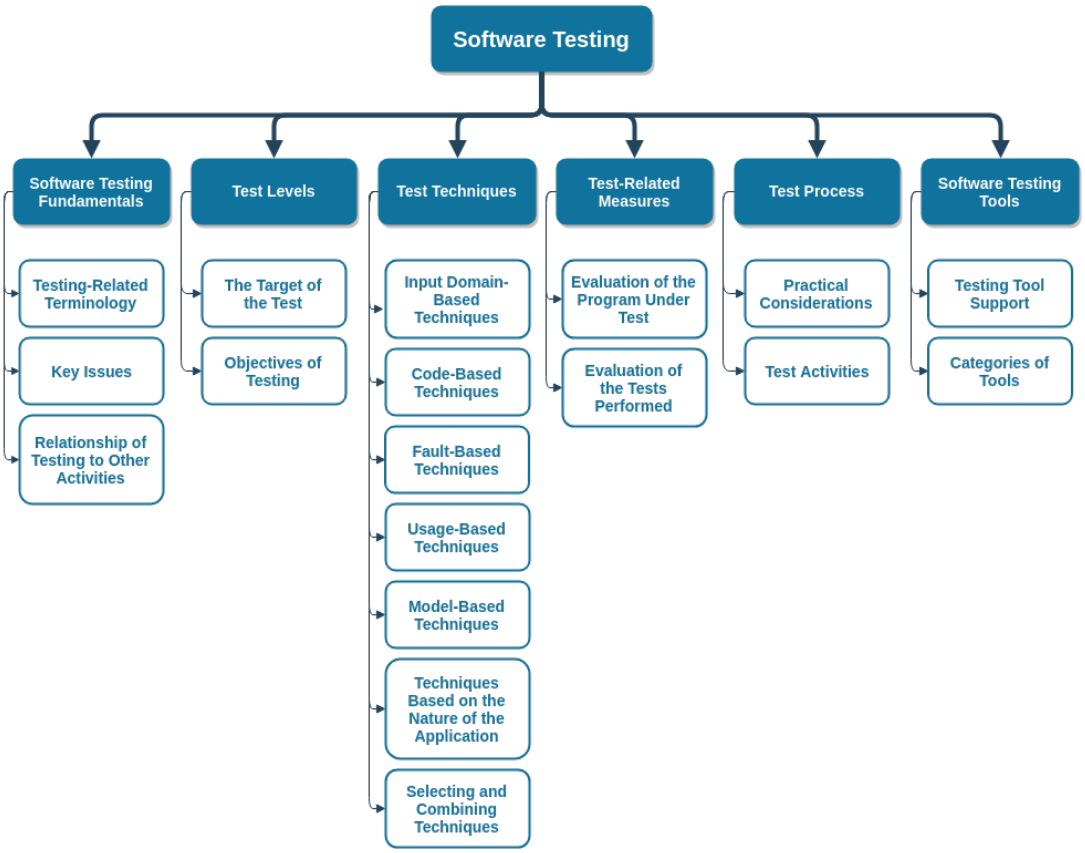


Figure 1: The hierarchy of the topics for the Software Testing Knowledge Area based on the SWEBOK guide [23].

2.2.2 Testing in a software development process

In a software development process the testing activities are usually regarded just as important as the development activities. This can be represented by the V-model [24]: There's two branches in the symbol "V". The left branch stands for the development process while the right branch represents the testing process. The V-model is divided into levels where each level has a development activity and a corresponding testing activity that verifies and validates it.

The V-model is illustrated in Figure 2.

The development activities as described by Spillner et al. [25] from up to down in the left branch are:

- Requirements definition
- Functional system design
- Technical system design
- Component specification
- Programming

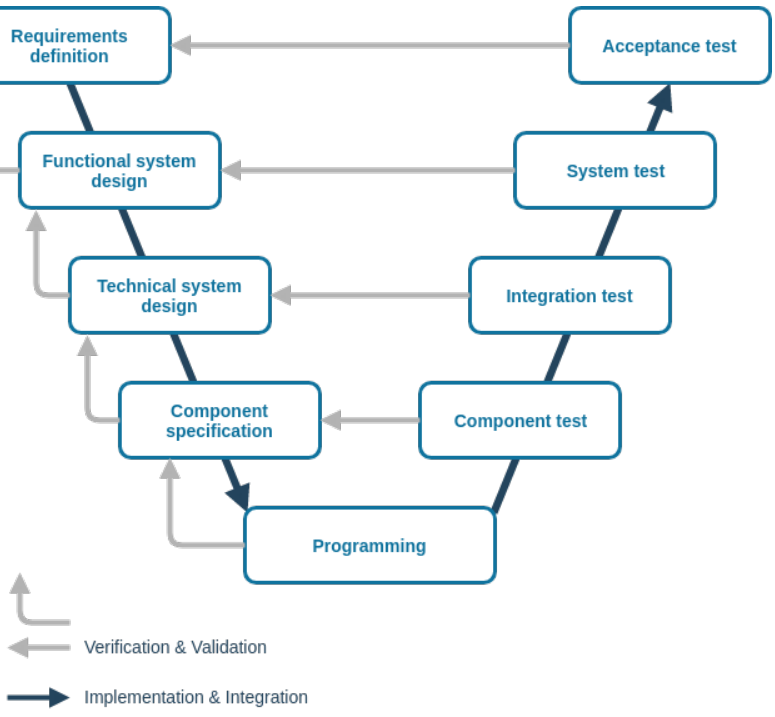


Figure 2: The V-model as described by Spillner et al. [25].

The testing activities to verify these development activities are described next.

The testing activity that verifies the component specification is **component testing**. Components can be, for example, modules, units and functions. Component tests verify that the implementations of these components behave as expected. Component testing is also known as unit testing.

Integration tests verify technical system design. They test that all the components work together correctly and that there are no bugs in the interactions.

In the next level of the development process there is functional system design that is verified by **system tests**. System tests make sure that the product meets the requirements specified when designing the functions of the system.

Acceptance testing is the final phase. They test that the requirements defined in the first phase of the development process are met and that the product is ready to be presented to the user.

Testing can be either dynamic or static: dynamic testing involves execution of the test object while static testing does not [22]. In dynamic testing, the tester works with the test object for example by giving it inputs and checking if the output is correct. Some dynamic testing can be done with automatic tools, such as automated tests. Tests libraries and frameforks, such as JUnit[1] for Java, are useful for this. Static testing can be done by reviews or tool-driven evaluation of the components of the test object.

Specifying the conditions and observing and recording the results are integral parts of testing software systems [13].

Regarding of when the tests should be written in a software development process, there are two approaches: in the test-first approach the tests are written before the source code and in the test-last approach they are written after the source code.

The test-first approach is a central element in test-driven development (TDD) [26], which is a popular software development process. In TDD the development cycle goes on in the following manner:

1. Add a test
2. Run all tests and see if the new test fails
3. Write code so that the test passes
4. Run tests
5. Refactor code
6. Repeat

It should be noted that TDD is a development process, not a testing technique.

[1] `https://junit.org/junit5/`

There has been some debate on whether it's better to test first or test last. The test-last approach is usually more intuitive and perhaps easier to adapt, but there are studies that suggest that it's actually better to test first. In their empirical study Nagappan et al. studied three software teams that used TDD and found out that their products contained less defects than products of teams that did not use TDD [27]. Furthermore, a systematic literature review conducted by Bissi et al., in which 1107 articles were gathered and 27 were studied comprehensively, suggests that using TDD practices can increase both internal and external software quality significantly [28].

However, Erdogmus et al. [29] and Janzen et al. [30], who both conducted studies using undergraduate students to compare the test-first and test-last approach, suggest that the approaches don't differ when measuring software quality. In their results the quality of the programs created by students who used the test-first approach was not shown to be better than those created by students who used the test-last approach, although the quality of the programs where the test-first approach was applied seemed to be more consistent. Then again, those who wrote the tests first wrote more tests, which led to being more productive. These results may seem contrary to those of Nagappan et al. and Bissi et al. but it's worthwhile to note that they studied the effects of using TDD, which is a development technique and concerns more than just writing the tests first, whereas comparing the orders in which the source code and tests can be written is a different matter.

A study by Fucci et al. suggests that the order in which the code and the tests are written doesn't matter as long as the process is iterative, the steps are small enough and the developing pace is steady [31].

2.2.3 The quality of software tests

Estimating the quality of software tests can be done in different ways. One of the most used metrics for assessment is thoroughness [32]. It can be measured by, for example, calculating the statement coverage or the branch coverage of the source code.

Statement coverage and branch coverage are calculated in the following manner:

- Statement coverage = number of executed statements / total number of statements
- Branch coverage = number of executed branches / total number of branches

They are explained more profoundly in Figure 3.

Code coverage metrics are used to ensure that every part of the code is tested. However, good code coverage does not necessarily indicate good test quality [33, 34]. Even if all source code is executed by the tests, the program may not behave as expected.

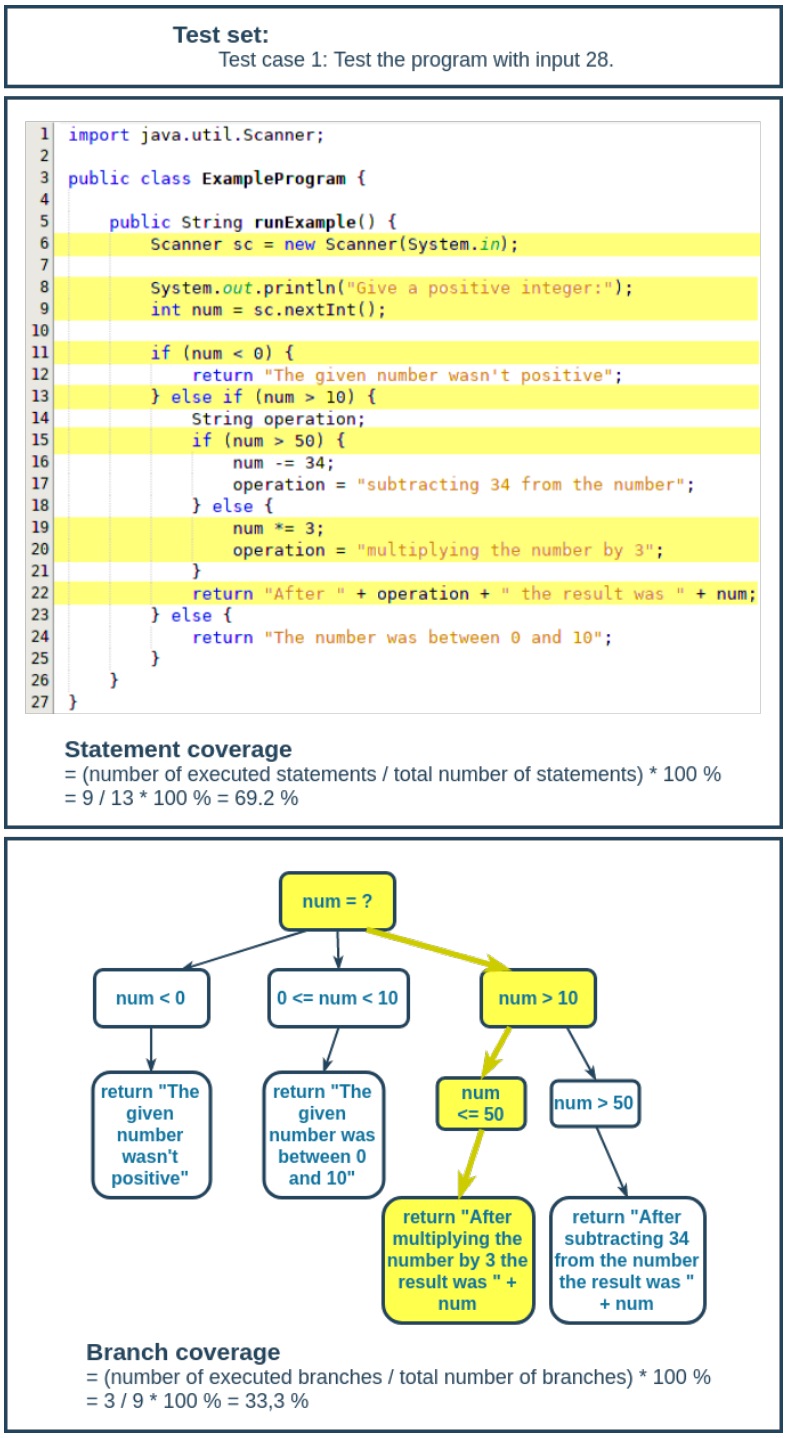


Figure 3: An example of how statement coverage and branch coverage are calculated. The example uses a test set of one test: "Test the program with input 28."
The executed statements in the source code are highlighted (lines 6, 8, 9, 11, 13, 15, 19, 20 and 22). The tree illustrates all the possible branches when executing the program. The path that the test set covers is highlighted.
The statement coverage of the test set is 69.2 % and the branch coverage is 33.3 %.

In their paper Aaltonen et al. compared code coverage metrics to mutation analysis. In mutation analysis artificial defects are inserted to the program and the tests are assessed by their ability to distinguish the mutated version from the original [33]. Their results suggest that using mutant analysis with code coverage metrics could be a more effective way to assess test quality than just calculating code coverage.

Goldwasser suggests an assessment strategy called all-pairs testing which can be used in an educational setting [35]. In this strategy students solve programming assignments as usual but provide a set of test cases with their solutions. Those tests are then run against every other student's program and their ability to reveal bugs is measured.

To study how well different test quality measures perform when assessing student-written programs, Edwards et al. conducted an experiment in a data structures course [36]. In their experiment they compared three test quality measures: all-pairs scores, mutant analysis and composite code coverage (counts how many of the methods, statements and branches of the code are executed by the tests). They found out that the all-pairs score correlated best with the students' tests' ability to reveal defects.

Tests can also be assessed by manual reviews. For example, planning and documenting the tests are both part of the testing process, but difficult to assess with automatic analysis.

To purely estimate one's testing skills is rather difficult, but the ability to test software well can be often related to one's programming skills.

2.3 Software testing in computer science education

Being such an essential part of ensuring software quality, it is easy to conclude that software testing should be taught in computer science education. But although the subject itself has been discussed in computer science literature at least since 1979 [37], it has not been emphasized in education as much as it might deserve [38].

Nevertheless, there has been several attempts to efficiently integrate software testing as a part of the computer science curriculum. In this subsection some of those attempts are discussed.

It seems the consensus is that the best practice for teaching software testing is to integrate it throughout the curriculum [39, 40, 5, 41, 42]. For example, Hilburn et al. suggest that the importance of software quality should be made clear to the students as early as in the first introductory programming courses so that it would get the emphasis it deserves [39].

Jones has put these ideas into use by introducing a holistic approach in which every student is taught the basics of software testing and is offered a lot of chances to get testing experience throughout the curriculum [40, 43]. This is done in order to teach the students the major principles of quality assurance which, according to Jones et al. [44], are the following:

- **Specification**, which is crucial to testing.
- **Premeditation**, a systematic design process that is required by testing.
- The testing process and the results must be **repeatable**.
- The testing process must be **accountable**.
- The testing process should be **economical**.

Jones suggests that when introducing this so called SPRAE (an acronym of the principles) framework in small doses, the students will internalize the testing concepts and practices easier.

The majority of the attempts to integrate testing into programming courses are done by including testing as a part of the programming exercises, usually by asking the students to give their own test cases for the programs they submit [35, 45, 5, 46, 41, 42, 2, 47]. Their programs are then evaluated by running the instructor given test cases as well as their own. The tests are also evaluated by checking that they pass and that they find the defects in the program.

Teaching testing is often linked with teaching Test-Driven Development (TDD): learning TDD is usually the goal, while learning testing is its side effect. This kind of approach has been taken by, for example, Edwards whose work [5] is rather similar to Jones' [40] with a few differences. In Edwards' approach, TDD is introduced to the students in the beginning of the course. After that they are required to put it into use when doing the programming exercises. The exercises are graded by using a service for automated testing called Web-CAT Grader. It is a web application that gives each submitted exercises three scores: test validity score (how many of the tests are accurate), test completeness score (how thoroughly do the tests cover the problem) and code correctness score (how correct the code is based on the students own tests). This service gives quick feedback of the quality of the solution and tests. Edwards used this approach in an undergraduate class called "Comparative Languages" in 2003. The students reported appreciation for using TDD and the number of bugs in their code decreased by 28 % per 1000 lines of code.

In another study, Edwards compared two courses, the first of which didn't use TDD nor the Web-CAT Grader to grade students exercises, while the other did [48]. The students in the latter course had 45 % less bugs per thousand lines of code.

Marrero et al. also experimented applying TDD practices in introductory programming assignments [46]. They attempted to make the idea of a class more easy to understand by first using it and writing tests for it and only after that implementing it. From their results it would seem that writing the tests before the source code might make the student's performance worse, not better. The authors address that there are a few possible explanations for that and that these results need more investigating.

Students do not always enjoy using TDD or writing tests [41, 49, 4, 45]. In their paper, Spacco et al. introduce a system for project submission and testing called

Marmoset that tries to tackle this problem [42]. It distributes the students the projects they are supposed to work on, including documentation, skeleton code and some test cases that are called public test cases. They define the basic functionality of the project. The students are also encouraged to write their own test cases for their code. After submitting the project, it is tested against the public tests. Marmoset also gives the students the opportunity to release test their project. Release tests are tests defined by the instructor and not shown to the student. Students are told the number of passed and failed release tests and the *names* of the first two failed release tests. Using this opportunity costs a release token and each student has a limited number of release tokens per project. These features are designed to make the students feel encouraged to write their own test cases and that way figure out why their project is failing.

Clegg et al. have attempted to make testing enjoyable and easier to learn by gamifying it [4]. They have created a mutation testing game called Code Defenders that teaches the players software testing concepts, such as code coverage, through gameplay.

Another testing-related game is the Dice Game used by Michael Bolton and James Bach on their lectures on software testing. While there doesn't seem to be any scientific research on how this game works, it is discussed in a few blog posts [50, 51, 52]. The game is led by a instructor who has a set of rules in mind. The players roll a set of dice after which the instructor tells the score. The players are supposed to find out why they got that score and what the rules are. While a person playing for the first time might first think that the score is based on the number of dots on the dice, the rules can be anything. The Dice Game is used in order to demonstrate some key skills a tester needs: analytical thinking, questioning one's assumptions, reverse engineering and thinking outside the box.

Although the consensus seems to be that software testing and TDD should be taught early on, it doesn't mean that it would be equally easy at all levels of education. By comparing 18 studies on implementing TDD in programming education, Desai et al. noticed that the results received from higher levels of education are more encouraging than those from the lower levels [53]. They suggest that rather than concentrating on teaching TDD only on the higher levels, perhaps there needs to be more effort in integrating TDD in introductory programming courses.

One possible way of doing so is Test-Driven Learning (TDL), first introduced by Janzen et al. [41]. Programming is often taught by showing examples of how the concepts are used and what the syntax looks like. Usually these examples aren't clear enough on how the element behaves. For example, let's assume that there's a Java class called Person but the implementation isn't shown to us, just this piece of code that demonstrates how to call one of the classes methods:

```
public void printInfo(Person person) {
    System.out.println(person.getInfo());
}
```

The user discovers what this method prints only after compiling and executing the code. However, we could demonstrate both the same information as above as well as the expected behaviour with a JUnit-test:

```
public void testPrintAge() {
    Person mary = new Person("Mary", 26, "software tester");
    assertEquals("Mary is a 26-year-old software tester.",
                 mary.getInfo());
}
```

The main idea of TDL is to teach by example. This is done by showing the students examples with automated tests and holding tests in high importance. Using a test-first approach in TDL comes rather naturally as it keeps the programmer focused on the item's interface and behaviour.

One problem with integrating testing into programming courses is that the curriculum is already full. Elbaum et al. present a tool called Bug Hunt that is designed to help educators with this struggle [54]. It is a web based tutorial in which the student goes through a number of lessons and uses test strategies presented in the lessons to find defects in programs. Students' progress is defined by how many defects their tests can spot and how they perform compared to other students. Every lesson consists of the objectives and an exercises. Bug Hunt aims to increase the students' knowledge of software testing little by little. By the time Elbaum et al. wrote this paper, over 400 students had used Bug Hunt. When asking the students if Bug Hunt had added significant value to the lecture material, 77 % of them "agreed" or "strongly agreed". 66 % "agreed" or "strongly agreed" that the tool had taught them useful concepts.

Desai et al. also tried to tackle this problem [2]. They attempted to integrate TDD into an introductory programming course's course material with minimal impact. Two student researchers revised the labs and projects of the course to include instructions on TDD and JUnit and developed test suites and new grading scripts and criteria for them. The course instructor had no experience with TDD and did not include it in his lectures, so he spent the minimal possible time integrating TDD into this course. At the end of the experiment he voluntarily chose to continue using the new lab and project materials in the future.

Christensen also participates in the discussion with his paper titled "Systematic Testing should not be a Topic in the Computer Science Curriculum" [55]. Regardless of the provoking title, the main point of the paper isn't actually that controversial: Christensen's opinion is that rather than regarding testing as an isolated topic in the computer science education, the educators should try to emphasize the significance of software quality. He put this idea into use on a programming course by setting some guidelines for the students such as "Make quality measurable" and "Make exercises a progression". The students then worked on their projects with quality as a priority. Many of them reported that the test-first approach was a great help when ensuring

that their programs were of good quality.

3 CrowdSorcerer

In this section CrowdSorcerer, the system for gathering programming assignments from students, is described in more depth.

3.1 Description

CrowdSorcerer [6] is a web-based tool designed to gather programming assessments created by students. It is used in introductory programming courses in University of Helsinki. As an educational tool, CrowdSorcerer is designed to be easily embedded directly into the course material. CrowdSorcerer is an open-source project and its frontend[2] and backend[3] can be found on Github.

CrowdSorcerer was created by the Agile Education Research Group (RAGE) (including the author) in the University of Helsinki for the following reasons. The first is educational purposes as teaching programming, reading code and learning to test programs. The second reason is to collect a pool of programming exercises to be used in programming courses in the future. The last reason for creating CrowdSorcerer was to do research: how do students create exercises of their own? How do they perceive various exercises made by other students?

3.2 Creating an assignment and the tests for it

In the course material there are written instructions for how to use the tool as well as a video. They explain all the phases in detail. First the student is given some kind of a topic, for example: "Create an assignment that asks the one doing the exercise to write a program that reads an input of type int from standard input, handles it with an if-else-statement and prints a String to standard output. Give test cases for the assignment. For a test case you must give a name, an assertion type, an input and the expected output for that input."

The student first writes instructions for the exercise in the assignment field of the tool. They can bold the text and use other similar enhancements by clicking the buttons above the text field. Then they provide the code template that will be shown to the user doing the exercise, as well as the model solution. Some boilerplate code has been given by the instructor. Boilerplate code means the lines of code that need to be included in the program. Those lines are greyed out and they can not be modified by the user. Next, the student will mark the lines that belong only to the model solution by clicking on the box next to the line number on the left of the

[2] `http://github.com/rage/crowdsorcerer`
[3] `http://github.com/rage/crowdsorceress`

source code field. The model solution lines will be highlighted in blue. One can also reset the source code field back to its original state by clicking the button “Reset source code field”. An example of an assignment and source code is shown in Figure 4 (the user interface of the tool is translated from Finnish to English for this thesis).

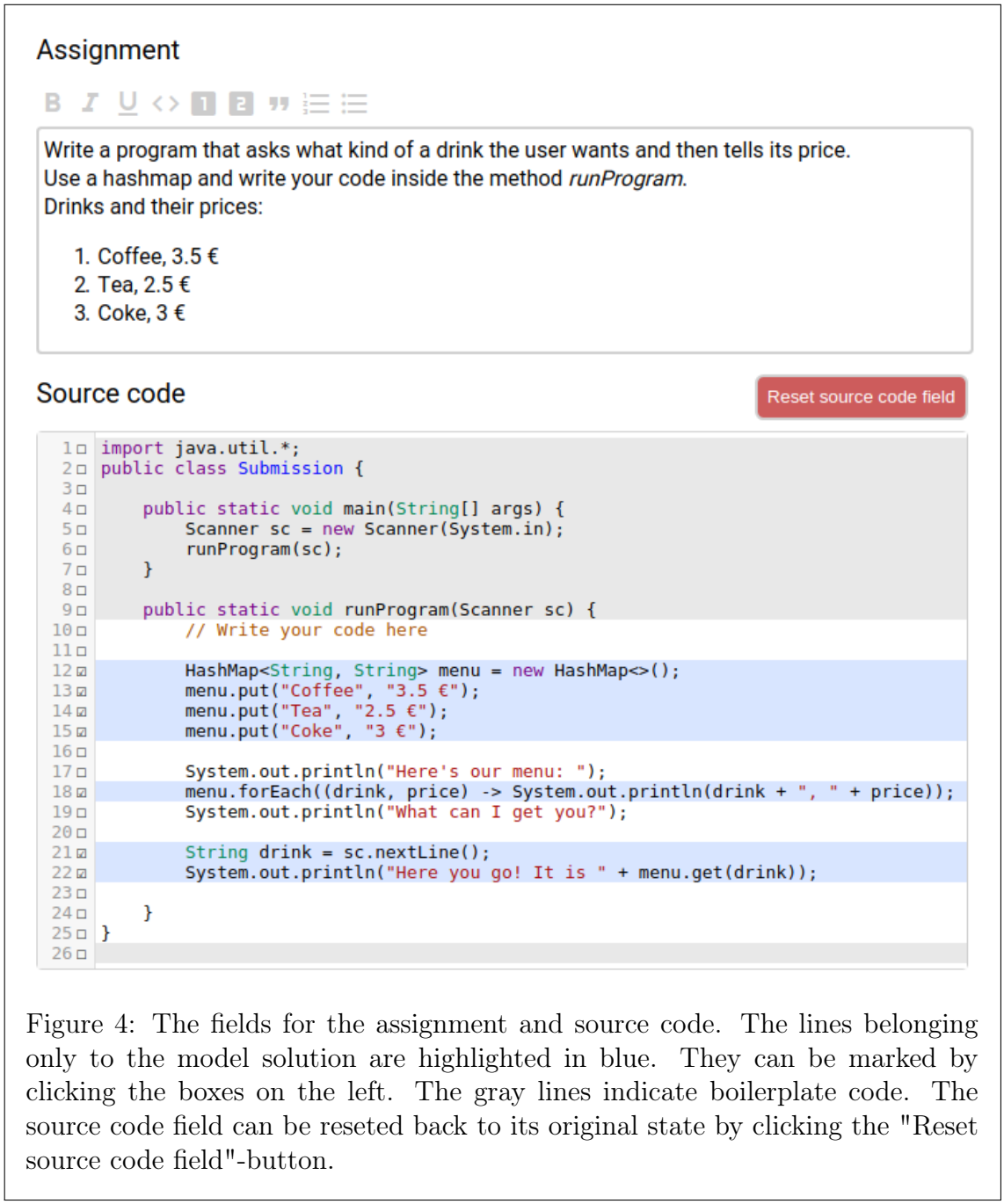


Figure 4: The fields for the assignment and source code. The lines belonging only to the model solution are highlighted in blue. They can be marked by clicking the boxes on the left. The gray lines indicate boilerplate code. The source code field can be reseted back to its original state by clicking the "Reset source code field"-button.

Next the student creates some tests for their exercise. CrowdSorcerer supports three ways for letting the student to create the tests.

The first way is giving the tests as pairs of inputs and outputs (see Figure 5). In the

user interface, the input field defines the input that is given to the program when the test is executed and the output is the expected output of the program after the execution.

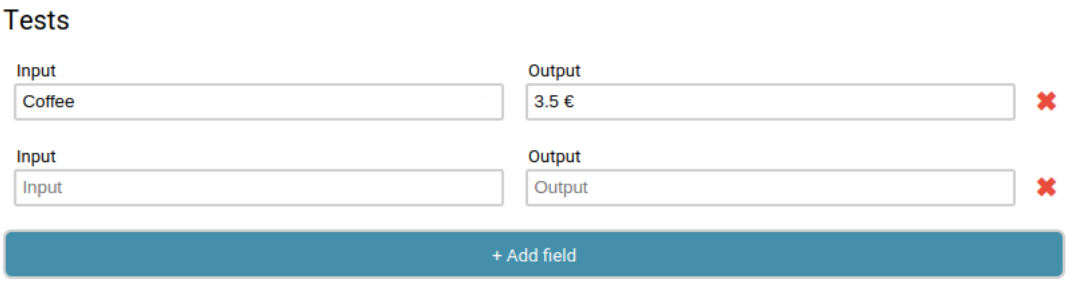


Figure 5: The first step when introducing tests in CrowdSorcerer is writing simple input-output tests. In this phase inputs and outputs are written in the given fields and the actual test code isn't shown to the user. At least one test case must be given. More test cases can be added by clicking the "Add field"-button and they can be deleted by clicking the red cross on the right.

The second way also consists of the student giving the input and output, but in addition to that they also give each test case a name and a type of the assertion (see Figure 6). Supported assertion types are "Contains", "Does not contain" and "Equals" and they define the relation of the user-given output and the output of the program: for example, if the type is "Does not contain", then the output of the program should not contain the output given by the user. In the exercises using this testing type the student will also see the test method generated from the name, type, input and output. The name, input and output are text fields whereas the assertion type is selected from a dropdown menu. Students can add more test fields and delete them, but at least one test case must be given.

The last possible method of creating tests lets the students write the whole test code themselves. This is done in the same way as writing the model solution: there is a text area where some boilerplate code is given. In this testing type three test methods are required.

CrowdSorcerer uses a Java library called edu-test-utils[4] in the test templates. The MockStdio class and its method getSysOut(), which reads the output of the program, are from that library.

The ways to create tests are summarized in Table 1 and illustrated in Figures 5, 6 and 7.

Before submitting the exercise the student gives it tags which are related to the exercise. Tags are added by typing them and pressing enter. The tool contains

[4] `https://github.com/testmycode/edu-test-utils`

Figure 6: In the second phase of creating tests with CrowdSorcerer the user gives each test a name, an input and the expected output and selects the assertion type from a dropdown list. The generated test method is shown below those fields. At least one test case is required.

some tags that are suggested if the user starts typing something similar. At least one tag is required. The field for tags is illustrated in Figure 8.

After clicking "Submit" an additional box opens where the student can preview the code template and the model solution separately and check that they are correct. Then they have to click "Submit" again. If they have missed a step in the assignment creation, e.g. left a field empty, the tool takes them back and shows the errors. Otherwise the exercise is sent to the server. The student is notified about the progression on the server with messages such as "Exercise saved to the database" and "Testing model solution" shown above a progress bar. If the code does not compile or the tests do not pass, the tool gives the user an error message.

The sequence diagram illustrating the assignment creation can be seen in Appendix 7.

The crowdsourced assignments are then reviewed by peer students. In the peer

Test code

```
1  import java.util.*;
2  import org.junit.*;
3  import fi.helsinki.cs.tmc.edutestutils.MockStdio;
4  import static org.junit.Assert.*;
5  import fi.helsinki.cs.tmc.edutestutils.Points;
6
7  @Points("01-11")
8
9  public class SubmissionTest {
10
11     @Rule
12     public MockStdio io = new MockStdio();
13
14     public SubmissionTest() {
15     }
16
17     @Test
18     public void testCoke() {
19         Submission.runProgram(new Scanner("Coke"));
20         String out = io.getSysOut();
21         System.out.println(out);
22         assertTrue(out.contains("Here you go! It is 3 €"));
23     }
24
25     @Test
26     public void TEST_NAME() {
27         fail();
28     }
29
30 }
```

Figure 7: The third and last possible way of creating tests with CrowdSorcerer is writing the actual test code. The base of the test class is given and the user is supposed to write a minimum of three test methods.

	Input-output-tests	Automatically generated test methods	Student-written test code
Users' input	Each test's input and expected output	Test name, assertion type, input and output	Actual test methods to a code template
Shown to user	Nothing. Test code is generated on the server side.	The test method generated from user's input	Whole test code
Minimum number of test cases	1	1	3

Table 1: A summary of the ways CrowdSorcerer supports creating tests for programming assignments.

reviewing phase, the students are usually given two assignments created by other students and one assignment that they have created themselves. They will analyze the assignment, the code template, the model solution and the test cases and review them. Some review statements are given by the instructor and the students use

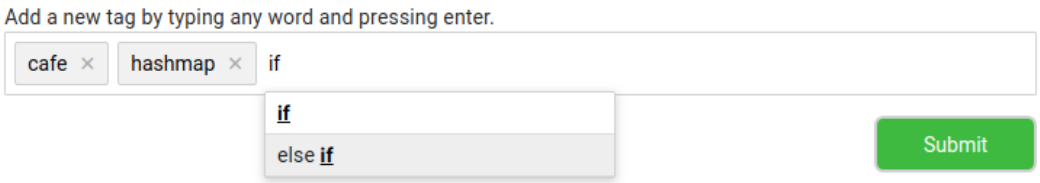


Figure 8: In CrowdSorcerer tags are added by typing any word, or words, and pressing enter. After pressing "Submit" the assignment is sent to the server where it is tested and saved.

a graphical Likert scale to give grades from 1 to 5 based on how well the exercise corresponds to those statements. Besides those pre-defined review statements, they give also written feedback and tags for the assignment.

3.3 Architecture

The architecture of CrowdSorcerer consists of three parts: a frontend, a backend and a sandbox server. The frontend is built with React and it is used to create the user interface that can be embedded to any online course materials. The frontend uses Redux to save local state. The backend is a Ruby on Rails application, used to store the data. The frontend uses a REST API provided by the backend to create the exercises and peer reviews. They also communicate with each other by broadcasting messages via a WebSocket connection. The sandbox is a part of the Test My Code system [56], which is a system that can be used to automatically test solutions to given programing assignments. CrowdSorcerer uses the sandbox to check that the exercises compile and that the tests pass. If the exercise passes the checks, it gets the status "finished", otherwise its status will be "error". The architecture is illustrated in Figure 9. The class diagram of the backend can be seen in Appendix 7.

The development of CrowdSorcerer started in summer 2017. The idea was to get a larger pool of programming assignments to use on programming courses as well as give the students more opportunities to practice on writing programs.

React was chosen as the framework for the frontend because it is simple and easy to learn. Ruby on Rails was chosen because it was familiar to the team and it is rather easy to create RESTful APIs with it.

The development started by creating the basic functionality, like the source code field, generating a Java project from the JSON data sent from the frontend to the backend, and sending the project to sandbox as an asynchronous job. Later some aspects have been improved: the user interface have been simplified and error messages have been clarified, for example. More possibilities of creating tests have also been added: in the beginning only input-output-tests were available. The motivation for adding more testing types was to see if introducing unit testing little

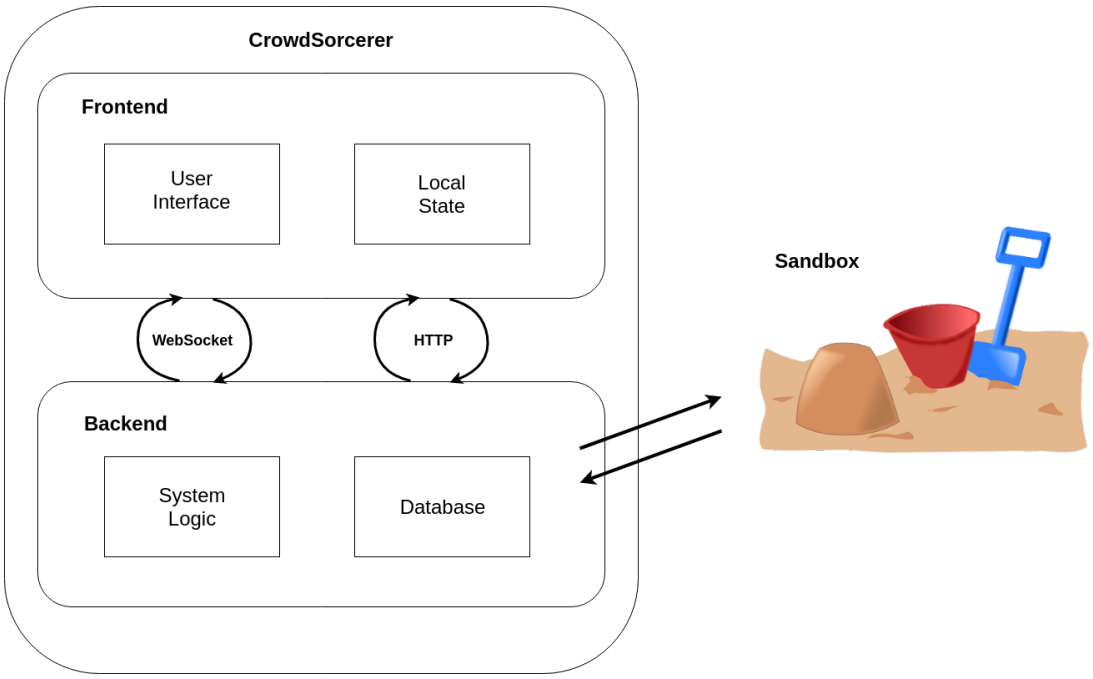


Figure 9: The architecture of CrowdSorcerer. The tool consists of a widget that can be embedded to a webpage, a backend for submissions, and a sandbox server that tests the exercises.

by little makes it easier to learn testing.

4 Methodology

In this section the research context and approach are introduced, including the research questions. The section also describes how CrowdSorcerer has been evaluated.

4.1 Research context

University of Helsinki offers two introductory programming courses that use Java as the programming language to be learnt and that are organized multiple times a year. The courses are called Introduction to Programming and Advanced Course in Programming, the latter of which is meant as a logical follow-up course to the former. The courses are organized as normal university courses for local students with lectures and on-campus tutoring, as well as MOOCs which are free for anyone to participate. Both courses use the same material and students complete the courses by doing exercises and taking the exams.

The material the courses use is an online material that, as a whole, consists of 14

weeks of educational content and exercises, starting from the basics of programming and ending with creating larger Java programs with graphical user interfaces, such as the game Snake.

Most of the exercises are traditional programming assignments, where the student is given an assignment and is supposed to write a program for it. In addition to that, the courses use CrowdSorcerer, which is described in Subsection 3.1.

The research for this thesis was conducted by gathering and analyzing the data from two courses, concentrating on the CrowdSorcerer assignments created by the students. The first of those courses was the Advanced Course in Programming organized as a normal university course in autumn 2018 and the second was Introduction to Programming organized as a MOOC in spring 2019. These courses will be referenced to as the advanced course and the introductory course later in this thesis. One could get a right to study Computer Science in University of Helsinki by performing the MOOC and its follow-up advanced course well enough (for additional details, see [57]).

4.2 Data collection

The assignments for which the students created their submissions varied in these two courses: some had instructor-given source code that could not be modified and the student's were only required to write a suitable assignment and give comprehensive test cases for it, whereas in other weeks the students created the whole assignments themselves.

The following is an example of the instructor-given source code, for which the students created the assignment and test cases. It was used as the source code on the third CrowdSorcerer instance in the Introductory Course (it is translated into English for this thesis).

```
import java.util.*;

public class Submission {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("How old are you? ");
        int ownAge = Integer.parseInt(sc.nextLine());
        System.out.println("Give someone else's name and age"
                           + "separated by a linechange: ");
        String otherName = sc.nextLine();
        int otherAge = Integer.parseInt(sc.nextLine());

        System.out.println(compareAges(ownAge,
```

```
                                        otherName,
                                        otherAge));
    }

    public static String compareAges(int ownAge,
                                     String otherName,
                                     int otherAge) {
        if (ownAge < otherAge) {
            return "You are younger than " + otherName;
        } else if (ownAge > otherAge) {
            return "You are older than " + otherName;
        } else {
            return "You are the same age.";
        }
    }
}
```

The first of those two courses, the advanced course, had a total of five instances of CrowdSorcerer. The first two of them were embedded in the material on week 1 of the course, the second two in week 3 and the last one in week 5.

The first two instances of CrowdSorcerer had instructor-given source code for which the students created an assignment and test cases. The students were required to create at least one test case. These instances used automatically generated test methods from user's input as a way to create tests (see the second column in Table 1).

The next two instances also had instructor-given source code. The last CrowdSorcerer instance required the student to create the whole assignment, including the model solution, themselves. These three instances used student-written test code as a way to create tests (see the third column in Table 1). Students had to write at least three test cases.

The advanced course also had a testing-related question in the final exam. The question introduced a program with a test-based user interface. It had a couple of methods. The student was able to execute the program and see how it worked. They then had to write unit tests for it. They were encouraged to think of how to test the program as extensively as possible and to make sure that the tests would tell what went wrong when executing them. Some examples of test cases were offered. The points gained from that exam question were also examined.

The latter of the two courses, the introductory course, had six instances of CrowdSorcerer.

The first two of those were on week 3 and had instructor-given source code for which the students wrote an assignment and test cases. The way to create tests in these instances was writing the inputs and seeing the automatically generated test

methods (see the second column Table 1). The minimum amount of test cases was 1.

The next two instances of CrowdSorcerer were on week 5. The source code was given again, and the students wrote the assignments and test cases. This time the students wrote the actual test code themselves (see the third column in Table 1). At least three test cases were required.

The last two were embedded in week 7. The students had to create all parts of the assignment themselves: the assignment description, the model solution and the tests. The tests were given as inputs for which the students could see the automatically generated test methods (see the second column Table 1). On week 7 the student's weren't shown whether the sandbox tests passed or not for their submission. That is, the exercises were marked as done and the students gained points from them if the submission process was successful, no matter of the outcome of the sandbox tests.

In the advanced course, one could get some bonus points by creating assignments with CrowdSorcerer. In the introductory course, the points gained from CrowdSorcerer were included in the total course points. The decision to include CrowdSorcerer points in the total points in the latter course was made in the hope that it would encourage students to create more assignments.

We have also gathered feedback of CrowdSorcerer. It was gathered from the students participating in the MOOC in spring 2019, either the Introduction to Programming -part or the Advanced Course in Programming -part, in spring 2019. The feedback form was filled on week 13, that is, the second to last week of the Advanced Course in Programming -part of the course. Not all the students in the first part of the MOOC had continued to study this far along and not all answering to this feedback form had studied the course from the beginning.

The form asked the students to give a minimum of 60 words of written feedback of CrowdSorcerer. They were encouraged to list pros and cons as well as give suggestions for improvement of the system.

4.3 Research questions and approach

The research questions for this thesis are the following.

- RQ1: How has software testing been taught in Computer Science education?
- RQ2: How are software testing skills measured?
- RQ3: Does creating assignments and tests with CrowdSorcerer have an impact on software testing skills?
- RQ4: How could CrowdSorcerer be improved?

Research questions RQ1 and RQ2 are answered in Section 2 by referring to research done earlier on software testing and Computer Science education. The search for appropriate studies was done by using keywords such as *software testing*, *automated testing*, *teaching software testing* and *software testing in computer science education*. The main database for finding articles was Scopus[5].

To answer RQ3 the tests created with CrowdSorcerer on two courses are analyzed. The analysis is done by gathering this data for each student on the courses: to which CrowdSorcerer instances they submitted an assignment, and how many tests they created for each assignment they submitted. In the Advanced Course the points the students got from the testing related question in the assignment are also collected. This data is then analyzed by calculating averages, correlations, and so on. The correlations were calculated using Spearman correlations.

RQ4 is studied by doing a qualitative analysis on the feedback gathered from students in the Introductory Course.

Research questions RQ3 and RQ4 are answered in the next section.

5 Results

To study the use of CrowdSorcerer and the tests created with it, the data from two courses was collected and analyzed. The first of those courses, the Advanced Course in Programming organized in autumn 2018, had 167 participants while the other one, Introduction to Programming organized in spring 2019, had 711 students who participated and took the exam. The courses that were chosen to be studied were organised separately, that is, the latter, as the introductory course, didn't have the same students as the first one.

Descriptive statistics of the use of CrowdSorcerer in those courses are covered in Subsections 5.1 and 5.2. Then, in Subsection 5.3, the correlation between creating tests with CrowdSorcerer and the performance in a testing related exam question is calculated for those who took part in the advanced course. Finally, in Subsection 5.4, students' feedback regarding CrowdSorcerer is analyzed and summarized.

5.1 Number of submitted exercises per prompt

The number of submitted exercises for each instance was calculated to see if there were any differences between them.

5.1.1 Advanced Course in Programming (autumn 2018)

In the advanced course, there were five instances of CrowdSorcerer. The first two instances were on week 1, had instructor-given source code, and the required mini-

[5] `http://scopus.com`

mum number for test cases was 1. The instances used automatically generated test methods from the user's inputs. For the first instance, 38 students submitted an exercise which is 22.8 % of all students participating in the course. 35 (92.1 %) were finished, i.e., they compiled and the tests passed. Students created 2.91 test cases for each finished exercise, on average. For the second instance, 29 exercises were submitted, thus 17.4 % of the students submitted an exercise. 26 (89.7 %) of them were finished and they had 3.42 test cases each, on average.

The next two instances were on week 3 and also had instructor-given source code. Test code was written by the students with a minimum of three test cases. The first of them had 18 submissions (10.8 % of all students), 13 (72.2 %) of which were finished. They had 3.08 test cases, on average. For the fourth instance 10 students submitted an exercise (6.0 % of all students). 9 (90.0 %) of them were finished and the average for test cases was 7.0.

The last instance was on week 5. In it the students had to write the whole assignment as well as the test code themselves. The minimum for test cases was 3. This instance got 10 submissions, so 6.0 % of students submitted an exercise. 7 of them passed the checks and they had an average of 3.57 test cases.

These numbers are also shown in Table 2.

CrowdSorcerer instance	1 (week 1)	2 (week 1)	3 (week 3)	4 (week 3)	5 (week 5)
Students who submitted an exercise (percentage of all participants)	38 (22.8 %)	29 (17.4 %)	18 (10.8 %)	10 (6.0 %)	10 (6.0 %)
Finished	35 (92.1 %)	26 (89.7 %)	13 (72.2 %)	9 (90.0 %)	7 (70.0 %)
Errored	3 (7.9 %)	3 (10.3 %)	5 (27.8 %)	1 (10.0 %)	3 (30.0 %)
Required test cases	1	1	3	3	3
Avarage number of tests per finished exercise	2.91	3.42	3.08	7.0	3.57

Table 2: Number of done exercises in the advanced course. 167 students took the course. The "Finished" row shows the number of exercises where the code compiled and the tests passed while the "Errored" row contains the number of the exercises for which one or both of those checks didn't pass.

5.1.2 Introduction to Programming (spring 2019)

There were six CrowdSorcerer instances in the introductory course. The first two were on week 3, had instructor-given source code and at least 1 test case was required. Tests were given as inputs for which the students could see the generated test method. 644 students (90.6 % of all participants) submitted an exercise for the first instance. 613 (95.2 %) of them were finished and they had 3.29 test cases, on average. The second instance got 604 submissions (85.0 % of all students submitted), 531 (87.9 %) of which passed the checks and had an average of 2.44 test cases.

The next two instances were on week 5, had given source-code and a minimum of 3 test cases. Test code was written by the students. The first of these, i.e., the third instance on the course got 560 submissions (78.8 % of students submitted). 502 (89.6 %) of them were finished and they had an average of 3.05 test cases. There were 500 submissions for the fourth instance, so 70.3 % of the students submitted an exercise for it. 470 (94.0 %) of them were finished and they had 3.17 test cases, on average.

The last two instances were on week 7. The students wrote the whole assignment themselves. Tests were given as inputs for which the test method was generated. At least 1 test case was required. The fifth instance on the course had 438 submissions, so 61.6 % of the participants submitted an exercise. 15 (3.4 %) of them passed the checks and they had 2.47 test cases, on average. The last instance got 346 submissions (48.7 % of the participants submitted an exercise), 14 (4.0 %) of which were finished and had an average of 2.43 test cases each.

5.2 Number of submitted exercises per student

The number of submitted exercises and the average of test cases per finished exercise was calculated for each student to see if those who submitted more exercises were also more likely to create more tests. For some assignments the minimum number of test cases was 1 while for other it was 3. This may have caused bias in the test counts. To avoid that, the test counts are calculated for each student and for each assignment using this method:

```
standardize_test_count(test_count, assignment_test_minimum):
    if test_count == 0 || assignment_test_minimum == 1:
        return test_count
    else:
        return test_count - 2
```

Using this method the results highlight how many extra test cases the students

CrowdSorcerer instance	1 (week 3)	2 (week 3)	3 (week 5)	4 (week 5)	5 (week 7)	6 (week 7)
Students who submitted an exercise (percentage of all participants)	644 (90.6 %)	604 (85.0 %)	560 (78.8 %)	500 (70.3 %)	438 (61.6 %)	346 (48.7 %)
Finished	613 (95.2 %)	531 (87.9 %)	502 (89.6 %)	470 (94.0 %)	15 (3.4 %)	14 (4.0 %)
Errored	31 (4.9 %)	73 (12.1 %)	58 (10.4 %)	30 (6.0 %)	423 (96.6 %)	332 (96.0 %)
Required test cases	1	1	3	3	1	1
Average number of tests per finished exercise	3.29	2.44	3.05	3.17	2.47	2.43

Table 3: Number of done exercises in the first part of the introductory course, for those students who took the exam (n = 711). The "Finished" row shows the number of exercises where the code compiled and the tests passed while the "Errored" row contains the number of the exercises for which one or both of those checks didn't pass.

created in addition to those that were required.

5.2.1 Advanced Course in Programming (autumn 2018)

There were five instances of CrowdSorcerer in this course's material.

Most of the students, 129 (77.3 %) of them, didn't submit any exercises and the average number of test cases for them was, naturally, 0. 9 students (5.4 % of all students) submitted 1 exercise with an average of 2.33 test cases. 19 students (11.4 % of all students) created 2 exercises and their average was 3.24 test cases per finished submission. 3 students (1.8 %) submitted 3 exercises, the average number of test cases was 2.78. Two students submitted 4 exercises with 2.38 test cases on average. 5 students created an exercise with CrowdSorcerer every time it was embedded in the material. The average number of test cases for them was 2.84.

These results can also be seen in Table 4.

Finished assignments	0	1	2	3	4	5
Students	129 (77.3 %)	9 (5.4 %)	19 (11.4 %)	3 (1.8 %)	2 (1.2 %)	5 (3.0 %)
Average number of tests	0	2.33	3.24	2.78	2.38	2.84

Table 4: Number of students who have done 0, 1, 2, 3, 4 and 5 finished assignments in the advanced course. The percentage in the parentheses indicates the percentage of all students on the course (n = 167). The average number of tests are calculated for finished assignments after standardizing the test counts using the method described in Subsection 5.2.

5.2.2 Introduction to Programming (spring 2019)

In the introductory course, there were 6 instances of CrowdSorcerer and the students received course points by submitting exercises for them.

58 students (8.2 % of all students) didn't submit any exercises. 59 (8.3 %) of the students submitted 1 exercise and the average number of test cases for them, after standardization, was 2.39. 2 exercises were submitted by 121 (17.0 %) students with 2.13 test cases on average. 72 students (10.1 % of all students) submitted 3 exercises and their average was 1.98 test cases per exercise. Most of the students, that is, 377 (53.0 %) of them created 4 exercises with CrowdSorcerer with an average of 2.07 test cases per exercise. Those 24 students (3.4 % of all students) who submitted 5 exercises had the highest average number of test cases: 2.83. There was no one who submitted an exercise for all six CrowdSorcerer instances on the course.

These numbers can also be seen in Table 5.

5.3 Performance in a testing-related exam question

In the advanced course there was a testing related question in the exam. 149 students (89.2 % of all students) answered it. The maximum points for that question was 15. The average points received from it was 10.36 with the standard deviation of 6.24.

When considering all students, we identified a statistically significant but small correlation between the number of exercises created with CrowdSorcerer and the points received from the testing related exam question ($r = 0.18$, $p = 0.02$). When limiting the analysis to only those students who answered the testing related question in the exam, there was no major difference in the correlation ($r = 0.19$, $p = 0.02$). Overall, the correlation is weak.

Finished assignments	0	1	2	3	4	5	6
Students	58 (8.2 %)	59 (8.3 %)	121 (17.0 %)	72 (10.1 %)	377 (53.0 %)	24 (3.4 %)	0 (0.0 %)
Average number of tests	0	2.39	2.13	1.98	2.07	2.83	0

Table 5: Number of students who have done 0, 1, 2, 3, 4, 5 and 6 finished assignments in the introductory course. The percentage in the parentheses indicates the percentage of all students on the course (n = 711). The average number of tests are calculated for finished assignments after standardizing the test counts using the method described in Subsection 5.2.

The correlation between the number of created test cases for the crowdsourced exercises and the points received from the testing related exam question was also calculated. The correlation was $r = 0.19$ with the p-value of 0.02. When calculated over those who answered the question the correlation was $r = 0.20$ and the p-value was 0.02. When also standardizing the test case counts using the method described in Subsection 5.2, the correlation was $r = 0.19$ with the p-value 0.02. Accordingly, there is a weak statistically significant correlation also between these variables.

5.4 Students' feedback on CrowdSorcerer

Feedback of CrowdSorcerer was gathered from the students who encountered the tool in spring 2019. The feedback form was a free text field where the students were asked to list pros and cons and give suggestions for improvement. A minimum of 60 words was required in order to gain a course point.

We randomly selected 100 feedback comments from the total number of 644 comments for more careful analysis.

Many of the students (21) thought that crowdsourcing programming assignments from the students and teaching testing while doing that is a good idea. 9 students felt that they had learned something by using CrowdSorcerer while 11 students thought that it hadn't been educational at all. 15 mentioned that especially the peer reviewing side worked well and 8 students liked the way CrowdSorcerer introduced testing to them.

> "The idea is really great and it has been beneficial to go through testing in practice. I don't think it has usually really been covered (at least in those introductory programming courses that I have taken) even though it's probably important in actual programming. The actual implementation of CrowdSorcerer isn't that great, and neither are the instructions.

> Ever since the first exercises it has been difficult for me to understand what was wanted. The use of CrowdSorcerer is easy, per se, but I would have needed a little more help and guidance for the actual exercises. In the latter exercises there was problems at least with Firefox as CrowdSorcerer crashed after submitting an exercise that didn't pass and would start working only after refreshing the window which then disposed all that was done until then. It would have been worthwhile to do the exercises on the computer first and only paste them to the browser, but at least I lost the written exercises more than once. That didn't really motivate creating more complicated exercises."

Still, many students thought that CrowdSorcerer was too difficult and slow to use. 45 students mentioned the crashing of CrowdSorcerer that happened often. Many (14) were frustrated with how slowly sandbox run the tests for their submissions and a lot of students (30) especially complained about how, when sandbox was finally done, the error messages were difficult to understand. The following is an example of a comment describing these problems (translated into English):

> "Creating exercises and especially tests is very confusing and unclear. When creating tests I don't even really know what should be done and in the rare case of trying to submit them they don't pass. And the suggestions for fixes are very unclear and once one succeeds to fix one thing, another is soon broken and one gets to fix things again. The purpose of tags has also remained unclear to me. The system also works slowly. Peer reviewing the exercises is easy and clear."

The user interface was described as complicated and stiff by 20 students but some also liked it: 10 students felt that it was simple and easy to use. Some suggestions for improvement were the possibility to enlarge the CrowdSorcerer-window and to make the progression bar appear lower in the tool since now it was difficult to find when one was scrolled to the bottom of the page.

One reason for the difficulties when using CrowdSorcerer was that the instructions were hard to understand: 25 students mentioned this. The separation between model solution and template as well as the purpose of tags were described unclear. Some thought that examples and videos on how to use CrowdSorcerer might have been helpful. Only one person wrote that they thought the instructions were clear.

Four students felt like they didn't learn any testing using CrowdSourcerer and hoped that more testing was covered in the actual course material.

There were 40 students who thought that the system just wasn't the best for this kind of purpose. They had for the most part created their assignments using an IDE and then copy-pasted it into CrowdSorcerer. They were wondering why the submission process couldn't also be done in the IDE when all the other exercises in the course could be done and submitted using the IDE. The text area in CrowdSorcerer, where the code was written, was unintuitive and difficult to use since it

lacked, among others, the key bindings and syntax checking the students were used to when programming using an IDE.

6 Discussion

In this section the results are investigated in a more detailed manner. Some proposals for improvement of CrowdSorcerer are also presented.

6.1 Experiences from using CrowdSorcerer

CrowdSorcerer was first deployed in autumn 2017. It has been used in the introductory programming courses of University of Helsinki for two years (autumn 2017, spring and autumn 2018, spring 2019).

CrowdSorcerer has also been a subject for research [10, 6, 11].

The traditional way to learn programming has been to read the material and do the exercises. CrowdSorcerer offers a different approach. It can be beneficial to think about what programs could be and to use one's imagination. The fact that someone else will read your code might encourage you to write better code. And reading and analysing other students' assignments and code may give a better understanding of what is good code.

Unfortunately, CrowdSorcerer has a number of weaknesses that complicate reaching the mentioned positive effects.

One of the main problems of the system is the slowness and unreliability of the test server. The TMC sandbox has been overloaded by the submissions and sometimes the submitting process hasn't completed at all. Or, at the very least, it has taken a very long time.

In the seventh week of the introductory course there was an attempt to overcome the difficulties to submit exercises by accepting all submissions and granting students points for their exercises whether they actually passed the tests or not. The actual results weren't shown to the user, only a notification that the submission had passed. This had an effect on the number of finished exercises for the fifth and sixth CrowdSorcerer, as can be seen in Table 3: 15 (3.4 % of all submissions) exercises submitted for the fifth instance and 14 (4.0 %) got the status "Finished".

The system also received plenty of negative feedback from the students, as the results from the analysis of feedback gathered from the students shows (see Subsection 5.4). The slowness and crashing of the system that occurred frequently frustrated the students. They also found the system rather difficult and complicated to use and the instructions and error messages weren't much of a help with that.

Some possible explanation for finding the system complicated is that writing code using a web-based interface can be much more difficult than using an integrated development environment (IDE) such as NetBeans, the use of which is encouraged

and supported by the courses. Although web-based applications are popular and a lot of things concerning every-day-life can be done using them, an actual IDE has numerous useful features that the students, completing their programming assignments with it, are used to. Those features include, for example, code completion and automatic error checking. Many web-based application for writing code lack those. In fact, in their feedback many students mentioned that they actually first wrote their code using NetBeans. They then copy-pasted the code to CrowdSorcerer, only using CrowdSorcerer for the submitting process.

The user interface was also criticized for not being clear and intuitive enough. The instructions should perhaps have been more explicit on how to use the tool, and the tool itself could probably be easier to use, so that it wouldn't need so many instructions. The text field where the code is written could be wider. The problem for this is that CrowdSorcerer is supposed to be embedded on the course material, so the layout of the material limits this. But perhaps one solution would be to make CrowdSorcerer an entire system of its own, or that it would open in a new window.

6.2 Number of created exercises

Creating exercises with CrowdSorcerer wasn't very popular in the Advanced Course. As Table 2 shows, the number of submitted exercises for each CrowdSorcerer instance was small. In the course, the largest number of exercises was submitted for the first instance of CrowdSorcerer, for which 22.8 % of the students created a submission. Towards the end of the course the number of submissions got even lower: only 10 students, 6.0 % of all students, submitted an exercise for the last CrowdSorcerer instance.

Table 4 shows that the exercises were mostly submitted by a small subset of the students as only 22.8 % of the students submitted any exercises on the course. Only 3.0 % of all students created a submission for each CrowdSorcerer instance.

One possible reason for why so few students submitted exercises is that the objectives of CrowdSorcerer were not clear enough. The idea of learning testing by creating programming assignments may have been difficult to grasp. The introductions for the use of CrowdSorcerer were also rather difficult and the tool didn't always work very well, possibly making submitting exercises frustrating. And since one did not necessarily need the bonus points given from those submissions, making them may have felt unavailing.

The corresponding results for the other course, the introductory course, show that more students submitted exercises for each CrowdSorcerer instance than in the advanced course (see Table 3). The highest percentage of students that submitted an exercise was for the first instance. Almost all students on the course created a submission for that. The lowest percentage of students who created a submission was the last instance but still almost half of the students created a submission.

Only 8.2 % of the students didn't create any submissions on the course (see Table

5). This also shows that creating submissions for CrowdSorcerer was more common in the Introductory Course than in the Advanced Course.

More people submitting exercises in the introductory course is probably due to the fact that the points gained from CrowdSorcerer in this course were included in the total course points, while in the advanced course one could only get some bonus points, so creating exercises wasn't as necessary to get a good grade from the course. For this reason creating submissions might have felt more rewarding in the introductory course.

6.3 Creating tests for the exercises

Although creating exercises wasn't very popular in the advanced course, the students who did submit exercises didn't seem to avoid creating tests. As can be seen in Table 2, in the first two CrowdSorcerer instances in the course the minimum number of test cases that were required was 1 but the average numbers of tests per exercise were 2.91 and 3.42, respectively. In the last three instances, where the minimum number of test cases that were required was 3, the average numbers of extra test cases weren't that high. For the fourth instance the average number was 7 but that could be explained by the fact that the source code provided by the instructor had multiple if-statements and one student of the 9 who submitted an exercise for the instance wrote a test case for each branch, therefore the data is slightly skewed.

When examining the average numbers of extra test cases for groups of students having finished 1, 2, 3, 4 or 5 assignments in the advanced course, clear differences can not be found (see Table 4). In other words, those who submitted more exercises didn't create more extra test cases for them than those who submitted less exercises. In fact, the students who created two exercises during the course created the most extra test cases: 3.24.

In the introductory course the students also usually created more tests for the assignments than what was required, as can be seen in Table 3. Similarly to the advanced course, students who created more exercises in the introductory course didn't create that much many extra test cases than those who submitted less exercises (see Table 5). The group of those students who submitted 5 exercises created 2.83 extra test cases, on average, but those who submitted only 1 exercise created 2.39 test cases, which is just 0.44 test cases less.

These results can potentially be explained by the fact that many of the submissions' source codes included basic statements such as if-else-statements for which one can easily make up test cases: make one test case for each branch. These kinds of test cases are probably easier to come up with than, say, testing that the program handles exceptions correctly.

6.4 Learning testing with CrowdSorcerer

To investigate if creating exercises with CrowdSorcerer could teach students automated testing the results for a testing-related question in the exam for the advanced course was studied in Subsection 5.3. Most of the students answered the question.

The correlation between the number of exercises created with CrowdSorcerer and the points received from the testing related question, calculated first for all students and then for those who answered the exam question, were positive but statistically weak. This was also the case for the correlation between the number of created test cases and the points from the testing related question.

Thus, although the connection between how many submissions and tests a student created and the performance in the testing related exam question is weak, there is still a connection.

There is also a possibility that those who used CrowdSorcerer more and created more exercises also spent more time studying on the course altogether. That could be the reason for them scoring better. A possibility of selection bias exists, too, since the course instructor didn't partition the students into groups where one group would be obligated to create exercises and the other wouldn't use the tool at all.

Only four students out of the hundred examined thought that they did not learn testing at all through CrowdSorcerer. This could indicate that if efforts are concentrated on making the use of the tool easier to use and more reliable, this method of teaching testing could very well be viable.

6.5 Limitations

CrowdSorcerer has been in use for two years. During that time it has gone through numerous modifications and bug fixes. For example, the test creation feature has been developed to include multiple options instead of just the simple input-output tests. Those can be seen in Table 1.

The development-deployment cycle has been rather rapid which has meant that not all error fixes have been perfect. For example, in the introductory course in 2019 it was decided that since many students had trouble submitting their exercises due to the slowness of the system, all submissions were accepted in the last week, regardless of their correctness. The students were shown a message that the submission passed. This led to a situation where only around 4 % of the exercises got the status Finished and the rest had an error of some kind (see Table 3). Thus, the results from the seventh week are rather skewed.

During development there has also been other bugs that have had an impact on the submitted exercises. At one point in time, the system didn't save the test codes of the submissions anywhere, which made it seem like all tests passed every time. It took a couple of weeks before this bug was noticed.

The results derived from this study lack of external validity as the research setting

was more about gaining experience from using CrowdSorcerer rather than actually examining properly if creating programming assignments led to writing better unit tests.

When considering the results gained from comparing students' performance in the testing related exam question in the advanced course and the number of programming assignments created with CrowdSorcerer there is a possibility for selection bias. CrowdSorcerer was usually at the end of the week. That may have led to a situation where only the most active students submitted exercises. Then again, the students practiced testing in other exercises during the course, not only creating them with CrowdSorcerer. That is, CrowdSorcerer was not the only way that the students may have learnt testing.

There are multiple ways for future work on the subject. For example, more research could be done on how to teach software testing. CrowdSorcerer could also be further developed. Future work is discussed in more detail in Section 7.

7 Conclusions and future work

This thesis consists of experiences of using CrowdSorcerer to teach automated testing. The study was done by analysing the submissions from two programming courses in University of Helsinki organized in 2018 and 2019.

The research questions and their answers are the following.

- **RQ1: How has software testing been taught in Computer Science education?**
  We analyzed how software testing has been taught in computer science education by reviewing literature on the topic. Based on the literature, multiple researchers have studied ways to teach testing. However, it seems that there is no consensus on whether there is a "best way" of integrating testing into the curriculum and what it is, if there was one. Similarly, there seems to be no consensus on when in the curriculum teachers should start teaching testing.

- **RQ2: How are software testing skills measured?**
  This was studied by exploring previous research on software testing. No clear answer was formed for how to measure testing skills but those skills may be linked with one's programming skills.

- **RQ3: Does creating assignments and tests with CrowdSorcerer have an impact on software testing skills?**
  We studied whether creating assignments and tests with CrowdSorcerer benefits software testing skills by analyzing correlations between the use of the tool and outcomes in an exam setting, where students were expected to write tests. There was a weak statistically positive correlation indicating that the use of the system may help learning the topic. However, we acknowledge a selection bias as CrowdSorcerer was always placed at the end of each week's material.

- **RQ4: How could CrowdSorcerer be improved?**
  The answer for this research question was obtained by analysing students' feedback. CrowdSorcerer could be improved in a lot of ways. The main improvements would be to make the system more reliable and faster and the user interface clearer and more intuitive. The improvement suggestions given by students considered mainly the system and its reliability, not how it attempted to teach testing.

CrowdSorcerer as a system would benefit from further development. Some ideas for that could be chosen from the students' feedback (see Subsection 5.4). For example, the user interface could be redesigned to be more intuitive to use. One possibility is that the crowdsourced assignments could be written in an IDE and submitted in the same way as the "normal" exercises.

In case of deciding to use the current UI, some improvements could be deployed. The area where the code is written could be wider, for instance, and it could have features similar to NetBeans, such as code completion and refactoring. The progress bar could be at the bottom of the tool, not in the middle as one may not realize that they have to scroll there. The system should also be faster and more reliable, and the error messages should be clearer.

A more precise research setting would be favourable when doing more research on teaching testing. Students could be partitioned into groups where one group was obligated to create assignments and the other didn't. Testing skills should also be measured some way, like in a well planned exam question.

Not placing CrowdSorcerer at the end of the week could be preferable. There are a lot of exercises during each week and students might not have time to create exercises after that. Creating a novel assignment can be rather time consuming.

The instructions should be clearer. Testing in general should be covered more in the course material. Now the students didn't gain the understanding of why testing is important. But, as covered in Section 2, it is difficult to decide how to include testing in the curriculum. More research could be done on the subject.

One possible way to study teaching testing with CrowdSorcerer is to study if introducing testing gradually (first the student gives only the inputs and outputs, then they see the generated code and then they write the test code themselves) has an impact on learning testing. It would be interesting to see if this approach makes it easier for the students to learn testing.

References

1 S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *SIGCSE Bull.*, vol. 36, p. 26–30, Mar. 2004.

2 C. Desai, D. S. Janzen, and J. Clements, "Implications of integrating test-driven

development into CS1/CS2 curricula," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, (New York, NY, USA), pp. 148–152, ACM, 2009.

3 R. Mugridge, "Challenges in teaching test driven development," in *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*, XP'03, (Berlin, Heidelberg), p. 410–413, Springer-Verlag, 2003.

4 B. S. Clegg, J. M. Rojas, and G. Fraser, "Teaching software testing concepts using a mutation testing game," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ICSE-SEET '17, (Piscataway, NJ, USA), pp. 33–36, IEEE Press, 2017.

5 S. H. Edwards, "Improving student performance by evaluating how well students test their own programs," *J. Educ. Resour. Comput.*, vol. 3, Sept. 2003.

6 N. Pirttinen, V. Kangas, I. Nikkarinen, H. Nygren, J. Leinonen, and A. Hellas, "Crowdsourcing programming assignments with CrowdSorcerer," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018, (New York, NY, USA), pp. 326–331, ACM, 2018.

7 P. Denny, A. Luxton-Reilly, and J. Hamer, "The peerwise system of student contributed assessment questions," in *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, (AUS), p. 69–74, Australian Computer Society, Inc., 2008.

8 P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Codewrite: Supporting student-driven practice of java," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, (New York, NY, USA), pp. 471–476, ACM, 2011.

9 A. Luxton-Reilly, B. Plimmer, and R. Sheehan, "Studysieve: A tool that supports constructive evaluation for free-response questions," in *Proceedings of the 11th International Conference of the NZ Chapter of the ACM Special Interest Group on Human-Computer Interaction*, CHINZ '10, (New York, NY, USA), p. 65–68, Association for Computing Machinery, 2010.

10 V. Kangas, N. Pirttinen, H. Nygren, J. Leinonen, and A. Hellas, "Does creating programming assignments with tests lead to improved performance in writing unit tests?," in *Proceedings of the ACM Conference on Global Computing Education*, CompEd '19, (New York, NY, USA), pp. 106–112, ACM, 2019.

11 N. Pirttinen, V. Kangas, H. Nygren, J. Leinonen, and A. Hellas, "Analysis of students' peer reviews to crowdsourced programming assignments," in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, (New York, NY, USA), pp. 21:1–21:5, ACM, 2018.

12 B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target," *IEEE Softw.*, vol. 13, pp. 12–21, Jan. 1996.

13 "IEEE standard for software and system test documentation," *IEEE Std 829-2008*, pp. 1–150, July 2008.

14 ISO/IEC, "ISO/IEC 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models," tech. rep., ISO/IEC, 2011.

15 G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. ACM*, vol. 21, pp. 760–768, Sept. 1978.

16 G. M. Weinberg and D. P. Freedman, "Reviews, walkthroughs, and inspections," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 68–72, Jan 1984.

17 K. E. Wiegers, *Peer Reviews in Software: A Practical Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

18 M. Shahin and M. A. Babar, "Improving the quality of architecture design through peer-reviews and recombination," in *Software Architecture* (D. Weyns, R. Mirandola, and I. Crnkovic, eds.), (Cham), p. 70–86, Springer International Publishing, 2015.

19 C. A. d. L. Salge and N. Berente, "Pair programming vs. solo programming: What do we know after 15 years of research?," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pp. 5398–5406, Jan 2016.

20 M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

21 ISTQB, "ISTQB glossary." `https://glossary.istqb.org`, 2015. Accessed: 2019-03-06.

22 ISTQB, "Foundation level syllabus." `https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html`, 2018. Accessed: 2019-06-07.

23 I. C. Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. Los Alamitos, CA, USA: IEEE Computer Society Press, 3rd ed., 2014.

24 K. Forsberg and H. Mooz, "The relationship of system engineering to the project cycle," *INCOSE International Symposium*, vol. 1, no. 1, pp. 57–65, 1991.

25 A. Spillner, T. Linz, and H. Schaefer, *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, 3rd ed., 2011.

26 Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

27 N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, pp. 289–302, Jun 2008.

28 W. Bissi, A. G. S. S. Neto, and M. C. F. P. Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review," *Information and Software Technology*, vol. 74, pp. 45 – 54, 2016.

29 H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, pp. 226–237, March 2005.

30 D. S. Janzen and H. Saiedian, "On the influence of test-driven development on software design," in *19th Conference on Software Engineering Education Training (CSEET'06)*, pp. 141–148, April 2006.

31 D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: Does it really matter to test-first or to test-last?," *IEEE Transactions on Software Engineering*, vol. 43, pp. 597–614, July 2017.

32 J. Wrenn, S. Krishnamurthi, and K. Fisler, "Who tests the testers?," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, (New York, NY, USA), pp. 51–59, ACM, 2018.

33 K. Aaltonen, P. Ihantola, and O. Seppälä, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, (New York, NY, USA), pp. 153–160, ACM, 2010.

34 Z. Shams, "Automatically assessing the quality of student-written tests," in *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, (New York, NY, USA), pp. 189–190, ACM, 2013.

35 M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, (New York, NY, USA), pp. 271–275, ACM, 2002.

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

37 G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

38 T. Shepard, M. Lamb, and D. Kelly, "More testing should be taught," *Commun. ACM*, vol. 44, pp. 103–108, June 2001.

39 T. B. Hilburn and M. Townhidnejad, "Software quality: A curriculum postscript?," *SIGCSE Bull.*, vol. 32, pp. 167–171, Mar. 2000.

40 E. L. Jones, "Software testing in the computer science curriculum – a holistic approach," in *Proceedings of the Australasian Conference on Computing Education*, ACSE '00, (New York, NY, USA), pp. 153–157, ACM, 2000.

41 D. S. Janzen and H. Saiedian, "Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum," *SIGCSE Bull.*, vol. 38, pp. 254–258, Mar. 2006.

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

43 E. L. Jones, "Integrating testing into the curriculum — arsenic in small doses," *SIGCSE Bull.*, vol. 33, pp. 337–341, Feb. 2001.

44 E. L. Jones and C. L. Chatmon, "A perspective on teaching software testing," *J. Comput. Sci. Coll.*, vol. 16, pp. 92–100, Mar. 2001.

45 E. G. Barriocanal, M.-A. S. Urbán, I. A. Cuevas, and P. D. Pérez, "An experience in integrating automated unit testing practices in an introductory programming course," *SIGCSE Bull.*, vol. 34, pp. 125–128, Dec. 2002.

46 W. Marrero and A. Settle, "Testing first: Emphasizing testing in early programming courses," in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, (New York, NY, USA), pp. 4–8, ACM, 2005.

47 V. K. Proulx, "Test-driven design for introductory oo programming," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, (New York, NY, USA), pp. 138–142, ACM, 2009.

48 S. H. Edwards, "Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance," in *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications*, EISTA'03, 2003.

49 D. Janzen and H. Saiedian, "Test-driven learning in early programming courses," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, (New York, NY, USA), pp. 532–536, ACM, 2008.

50 D. McMillan, "Dice game: Key life skills for testers," Jan 2011.

51 J. Rohrman, "Teaching software testing with games," Jul 2017.

52 J. McGuinness, "Dice, dice, baby. - the asos tech blog," Sep 2018.

53 C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *SIGCSE Bull.*, vol. 40, pp. 97–101, June 2008.

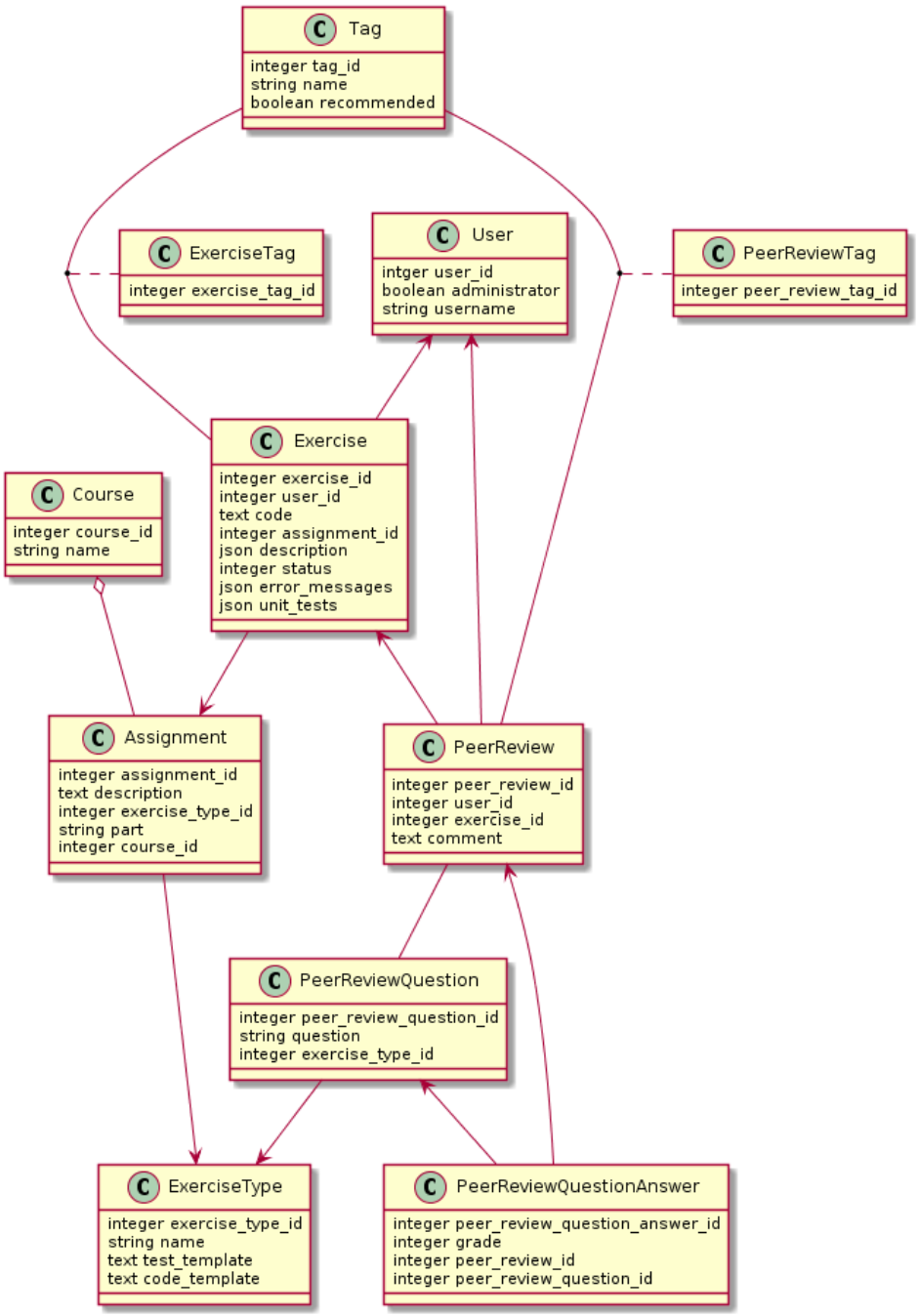
54 S. Elbaum, S. Person, J. Dokulil, and M. Jorde, "Bug hunt: Making early software testing lessons engaging and affordable," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 688–697, IEEE Computer Society, 2007.

55 H. B. Christensen, "Systematic testing should not be a topic in the computer science curriculum!," *SIGCSE Bull.*, vol. 35, pp. 7–10, June 2003.

56 A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel, "Scaffolding students' learning using Test My Code," in *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, (New York, NY, USA), pp. 117–122, ACM, 2013.

57 A. Vihavainen, M. Luukkainen, and J. Kurhila, "Mooc as semester-long entrance exam," in *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education*, SIGITE '13, (New York, NY, USA), p. 177–182, Association for Computing Machinery, 2013.

Appendix 1. Database schema of CrowdSorcerer



Appendix 2. The process of creating an assignment with CrowdSorcerer