Date of acceptance Grade

Instructor

On the Quality of Crowdsourced Programming Assignments

Nea Pirttinen

Helsinki June 2, 2020

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science	
Tekijä — Författare — Author Nea Pirttinen		
Työn nimi — Arbetets titel — Title On the Quality of Crowdsourced Programming Assignments		
Oppiaine — Läroämne — Subject Computer Science		
Työn laji — Arbetets art — Level Master's Thesis	Aika — Datum — Month and year June 2, 2020	Sivumäärä — Sidoantal — Number of pages 64 pages + 1 appendix pages

Tiivistelmä — Referat — Abstract

Crowdsourcing has been used in computer science education to alleviate the teachers' workload in creating course content, and as a learning and revision method for students through its use in educational systems. Tools that utilize crowdsourcing can act as a great way for students to further familiarize themselves with the course concepts, all while creating new content for their peers and future course iterations.

In this study, student-created programming assignments from the second week of an introductory Java programming course are examined alongside the peer reviews these assignments received. The quality of the assignments and the peer reviews is inspected, for example, through comparing the peer reviews with expert reviews using inter-rater reliability. The purpose of this study is to inspect what kinds of programming assignments novice students create, and whether the same novice students can act as reliable reviewers.

While it is not possible to draw definite conclusions from the results of this study due to limitations concerning the usability of the tool, the results seem to indicate that novice students are able to recognise differences in programming assignment quality, especially with sufficient guidance and well thought-out instructions.

ACM Computing Classification System (CCS):
Information systems → World Wide Web → Web applications → Crowdsourcing
Social and professional topics → Professional topics → Computing education programs → Computer science education

Avainsanat — Nyckelord — Keywords
crowdsourcing, computer science education, quality

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

The rapid rise of online education has provided new avenues for both lower and higher education in a variety of fields, computer science being no exception. Online education has been hailed as a solution for a variety of problems traditional schooling systems tend to have, such as expanding access across gender, racial, and financial borders [1]; alleviating capacity constraints [2]; and working with the growing demand of lifelong learning [1]. Computer science tends to dominate the lists of the most popular courses on the largest online education sites[1,2], with topics such as introductory programming, machine learning and algorithms.

Setting up and maintaining online courses brings its own set of challenges, different from the regular university lecture courses that are offered for a limited number of students each year. Model solutions tend to leak even during regular courses, and this problem is amplified with a larger group of unknown students. To make plagiarizing more difficult, exercises should be changed or updated between course iterations, which can bring the upkeep of an open online course to an exhausting level for an instructor who has to manage both online and lecture courses at the same time. Thus, alternative methods of creating exercises and course content are required.

Crowdsourcing is a practice of obtaining ideas, content or services from a large, usually open group of people. Since its popularization in the early 2000s [3], the hype around crowdsourcing has simmered down to a steadier level. With crowdsourcing efforts like reCAPTCHA [4] and Wikipedia[3], some Internet users do not realize that they are participating in or reaping the benefits of a crowdsourcing effort. Crowdsourcing not only produces content, whether that is an idea, a draft, or a fully working end product, but can also work as a game[4] or a learning experience for the worker [5]. This idea has been utilized in online education – as a part of their course work, students participate in crowdsourcing efforts, creating new materials and revising the course content at the same time.

Part of the contribution of this thesis is a system called CrowdSorcerer, developed by the Agile Education Research Group (RAGE), including the author, at the University of Helsinki. CrowdSorcerer [6] is a computer science education tool used for creating programming assignments. The tool can be embedded into online course materials, and the entire process of creating a full programming assignment can be done within the tool. Students create the assignments based on the specifics given by the instructor: the condition can be, for example, to create an assignment that uses conditional clauses. The end result is a programming assignment with a handout, a model solution, a code template and test cases. The assignments are automatically tested for compilation errors and run through the user-generated tests to make sure the program functions as intended.

[1]https://jwel.mit.edu/news/most-popular-moocs-focused-computer-science-self-help%C2%A0/
[2]https://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/
[3]https://www.wikipedia.org/
[4]https://crowdsource.google.com/imagelabeler/

CrowdSorcerer also has a peer reviewing functionality, which gives the students (and if need be, the instructors or assistants) the opportunity of grading each other's assignments. The reviewer has access to both the code template and the model solution, as well as the given tests. Reviews are conducted using an instructor-provided set of review statements, such as "The assignment handout is clear" and "The test cases are reasonable". The reviewer is also required to give written feedback.

The programming assignments created and collected with CrowdSorcerer could be used, for example, to help struggling students by giving them more small exercises covering the topic they have trouble with. However, since the tool is fairly new, the studies on its effect are far from comprehensive as of now, and more research into the created programming assignments is required before the assignment bank can be used in this manner.

The purpose of this thesis is to explore what kinds of programming assignments students on an introductory programming course create, and whether their peer reviews to the said assignments are reliable when compared to reviews done by an expert. If the peer reviews are deemed to be reliable enough and reasonable in quality, it would be possible to use a crowd of students to both create the assignments and review them, greatly alleviating the instructors' workload when it comes to creating short programming assignments. This thesis extends the work by Pirttinen et al. [7] on the analysis of the quality of the assignments created with CrowdSorcerer. The previous study found out that novice programmers are able to give as good reviews as more experienced students. This suggests that crowdsourcing the review process can produce fairly good quality assignment databases, which is also supported by previous studies on similar tools [8, 9].

Computer science education tools using crowdsourcing that have been developed prior to CrowdSorcerer include, for example, PeerWise [10], CodeWrite [11], and StudySieve [12]. These systems are described in Section 2.1.2, alongside other examples of crowdsourcing in computer science education. Parts of this thesis are based on previous work by the author and her colleagues [6, 7, 13, 14], and the specifics of these studies are described in detail when relevant to this study.

This thesis is organized as follows. Section 2 gives an overview to the relevant literature, namely crowdsourcing, software quality, and statistical methods. Section 3 gives a more detailed system description for CrowdSorcerer, and describes the research design and relevant data collection methods. Section 4 presents the results of the analysis, and Section 5 further discusses the implications of the results, as well as the limitations and future directions of this research. Finally, Section 6 summarizes the findings of this thesis.

2 Background

The following subsections introduce the background literature of this thesis, focusing heavily on crowdsourcing and code quality. Section 2.1 describes crowdsourcing generally, after which Sections 2.1.1 and 2.1.2 review its usage in online education and computer science education, respectively. Section 2.2 defines the concept of quality from computer science and software engineering point of view, and Section 2.2.1 further discusses quality from the computer science education point of view. Finally, Section 2.3 describes the statistical methods used in the data analysis of this thesis.

2.1 Crowdsourcing

Crowdsourcing is a method of obtaining services, ideas or goods from an open network of people, usually on the Internet. The term first appears in a 2006 article by Howe [3], and it was born as a portmanteau between the words *crowd* and *outsourcing* (employing workforce outside of one's own company to perform required tasks).

Howe's article discusses the idea of companies using the Internet to outsource their workload to a (usually very large) crowd of people. The difference between outsourcing and crowdsourcing is the party who is chosen to complete the task, as depicted in Figure 1. In outsourcing, the task is usually moved from one company to another, and the one requesting the favour pays salary for the one completing the task. In crowdsourcing, the requester is not as necessarily a company, but an individual, a collective or some other kind of organization. While the issuer can pay for those who meet the set standards of their crowdsourcing initiative, monetary payoffs are usually low or nonexistent.

Nowadays, crowdsourcing is a term that is very much associated with the Internet [15], but it is not necessary that the talent of the crowd is collected online. Idea or resource collection can happen in real-life as well [16]. During the last decade, the hype around crowdsourcing has grown to new heights, and the growth of social media platforms has made both advertising and participation easier. Thus, ever wider audiences can join the activity, and companies using crowdsourcing may profit hefty sums [17].

A well-known example of crowdsourcing is Wikipedia[5], a multilingual online encyclopedia that is maintained through user effort based open collaboration. Crowdsourcing is also commonly used for small, tedious tasks that can be performed simultaneously by a large crowd. This approach is used for example by Amazon Mechanical Turk[6], a website through which businesses can hire users to complete discrete tasks that cannot be done automatically or computationally (see Figure 2 for the basic workflow between requesters and workers).

Idea collection through crowdsourcing does not produce tangible results immediately, but acts as a way for companies to reach out to their customers and scout for potential future products and services. Many major companies, such as Coca-Cola,

[5]https://www.wikipedia.org/
[6]https://www.mturk.com/

Heineken and Threadless, have used crowdsourcing as an idea collection method to decide on new designs or products [18].

Implicit crowdsourcing is a form of crowdsourcing where it is not made obviously clear for the crowd participating in the activity that they are producing information for some party. One way of implicit crowdsourcing is reCAPTCHA, a system that is seemingly used to establish whether the user of a website is a computer or a human [4]. While the system does exactly that, for example through image identification, it can also help with things such as improving machine learning or map accuracy. For example, blurry images of Google Street View have been included in reCAPTCHA challenges, making the user identify the house number or street name not readable by computers.

As mentioned before, monetary payoffs for the user are rare, and even when used, the hourly rates are very low. Analysis of Amazon Mechanical Turk, one of the crowdsourcing services that uses a wage system for users who complete requested tasks, revealed that the workers earn a median hourly wage as low as approximately two US dollars [20]. According to the same analysis, only about 4% of the workers manage to earn more than 7.25 USD per hour (the US federal minimum wage), even though average requester pays more than 11 USD per hour. However, lower-paying requesters post tasks more frequently, and the calculations are affected by the time spent on not working on a task, that is, time spent on finding a suitable task, working on a task that gets rejected in the end, or tasks that are not finished for user-based reasons [20]. Despite the low wages, Amazon Mechanical Turk is still a very popular system for users who want to have some extra earnings: according to a population dynamics and demographic analysis from 2018, the system has more than 100,000 workers available at any given time, with around 2,000 actively working [21].

Since monetary payoffs are not usually the driving force for users to complete crowdsourcing tasks, companies and other requesters need other methods of sparking interest and maintaining motivation in users. On the user side, motivations for crowdsourcing can be intrinsic or extrinsic – actions driven by personal interests and emotions, or by external factors such as outcome, recognition or payoff [22]. Self-efficacy seems to play an important part on one's willingness to contribute to communal online activities in general [23], and the feeling of being able to help someone boosts intrinsic motivation [24]. On the other hand, extrinsic motivations such as recognition within a community and a sense of belonging act as ways of attracting new users and retaining existing ones.

Main concerns with crowdsourcing have to do with the end-product quality and the trustworthiness of the crowd. This is given – when crowdsourcing a task, the issuer cannot be entirely sure of the level of skill their crowd has. Strict checks on the expertise of the workers hinders the basic advantages of crowdsourcing over outsourcing, that is, flexibility, swiftness and ease of access. Thus, companies or individuals wanting to use crowdsourcing usually accept the additional workload of going through low-quality contributions. Contributions of low quality are not necessarily only due to varying levels of competence in the crowd, but also because of

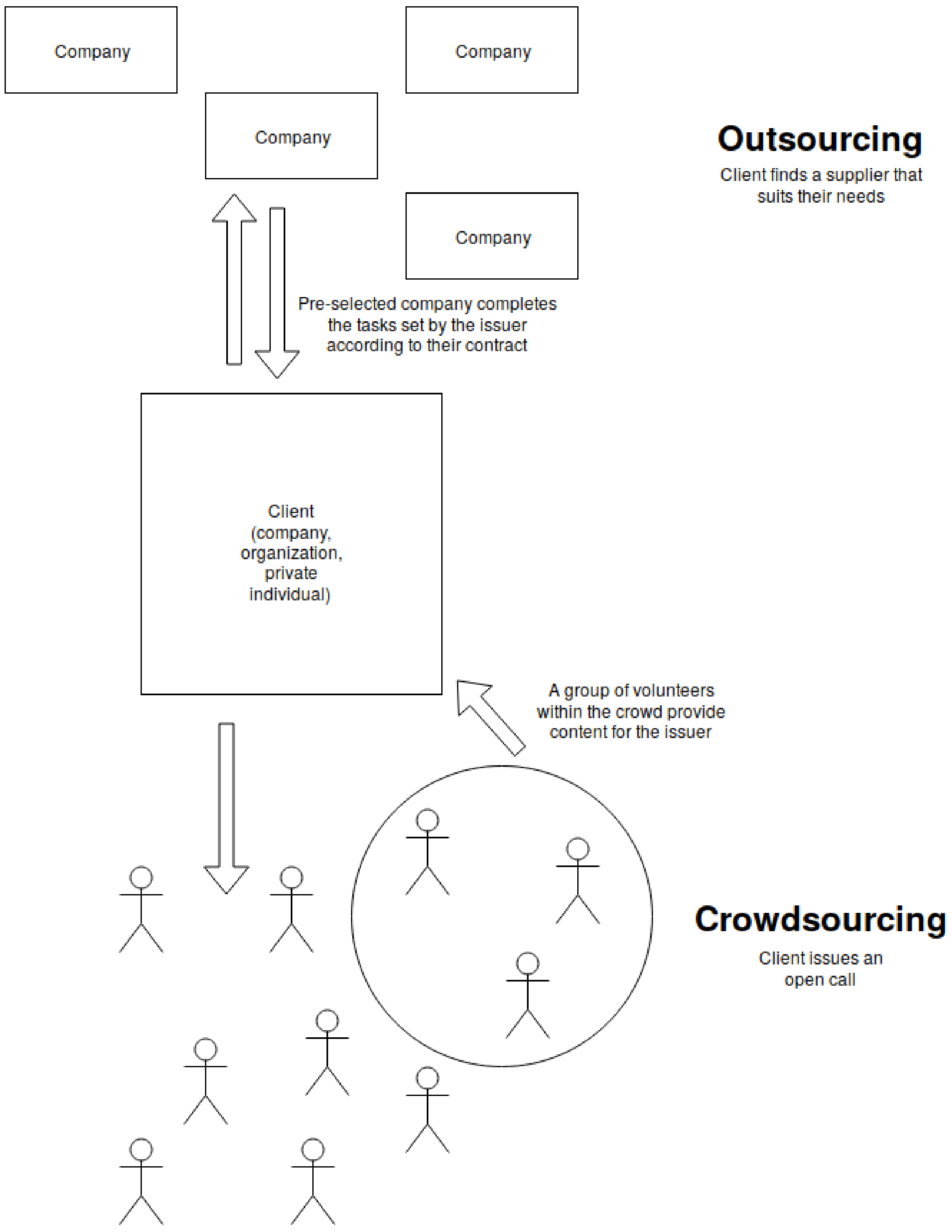

Company
Company
Company
Company
Outsourcing
Client finds a supplier that suits their needs
Pre-selected company completes the tasks set by the issuer according to their contract
Client (company, organization, private individual)
A group of volunteers within the crowd provide content for the issuer
Crowdsourcing
Client issues an open call

Figure 1: The differences of outsourcing and crowdsourcing. Figure adapted from [17].

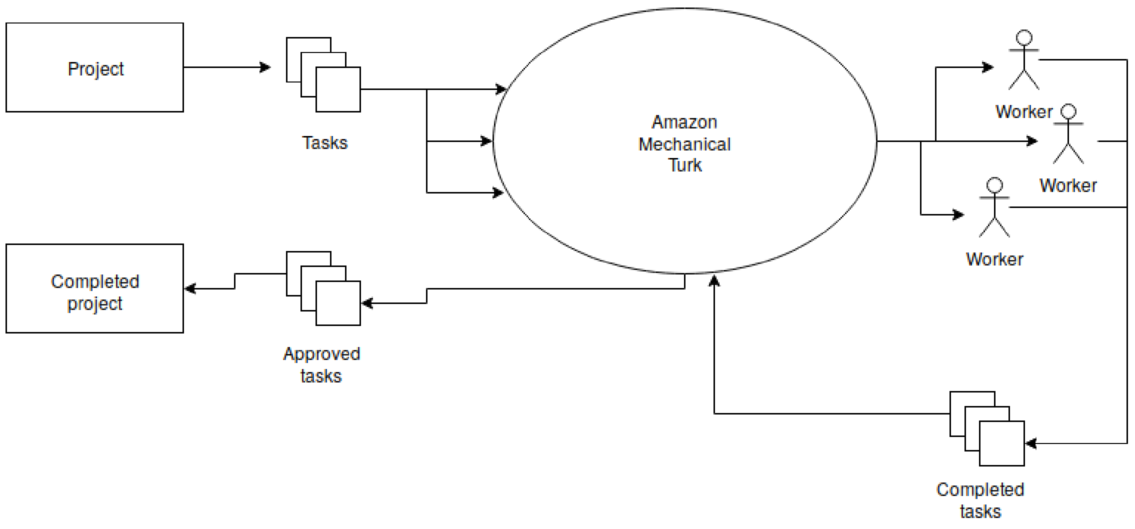


Figure 2: Amazon Mechanical Turk workflow. Figure adaped from [19].

intentional maliciousness, either for the person's own amusement or to hurt the company. The format in which tasks are given also affects the quality of crowdsourcing [25]: free-format seems to produce better quality contributions, even though the format is more difficult for the issuer, as the contributions need to be carefully checked before they can be used.

On the other hand, ethical issues such as using crowdsourcing as free workforce are also present. For example, Amazon Mechanical Turk has received some criticism [26, 27] for its low hourly wages [20], also presented above. However, one of the studies [26] also surveys the workers' attitudes, which indicate that most of the workers do not feel extorted with the wages they are receiving after being presented the realized hourly wages, and vast majority will continue with crowdsourcing activities in the future. Personal benefit in the form of recognition, experience, feelings of contentment from being able to help, or reciprocity outweigh the lack of monetary benefits.

Critics have also noted that crowds are not necessarily as uneven or lacking in their skills as is commonly believed [28]. Instead, many users participating in crowdsourcing activities are actually professionals or experts in the field, and take part willingly as a pastime, or for additional recognition and experience in the field. Thus, labeling crowdsourcing as an amateur-only activity undermines the expert-level contributions from professionals, and further reduces the incentive to pay proper salaries for the crowdsourcing activities where a payment would be suitable [28]. The prevalence of expert-level participants also means that even though the crowd might be uneven in skills, crowdsourcing activities are capable of producing good quality end products.

2.1.1 In online education

During the recent decade, online education has surged in popularity. Ranging from self-study materials to open university courses, online courses have been regarded as a vital part of making different levels of studies available for all. For example, Coursera[7] offers hundreds of free courses, topics ranging from particle physics to social studies.

Online education has become core part of computer science studies [29]. Especially MOOCs (massive open online courses) have become a very popular way to tackle courses [30, 31, 32], as they are readily available for a range of people of all ages, genders, and education levels, regardless of their physical location and monetary situation [1]. MOOCs on programming have even been used as an entrance exam to bachelor's studies in computer science [33, 34], and have been proposed to alleviate the lack of computer science related studies before university level [35]. In addition, MOOCs have also been used to teach various other computer science related topics beside programming, such as databases, algorithms and machine learning, and introducing teachers to teaching computer science [36, 37].

While ease of access and flexible scheduling help the participating students [29, 31], online education brings its own set of challenges, one of them being the sheer number of people attending the course at the same time. This means that one course instructor will not necessarily have the time for personal assistance, at least not very intensively, and many of the evaluating procedures require some level of automation [38]. Also, as the model solutions tend to leak to the Internet during the course, it would be beneficial to change the exercises at least partially between the course iterations to make straightforward plagiarizing more difficult. Changing exercises for every course iteration requires a massive pool of exercises and answers, which in turn takes time and effort to build and maintain. Such efforts might be infeasible for a course instructor who most likely teaches regular lecture courses and can not spend all of their time for creating new exercises.

One way of alleviating the pressure of creating new material and exercises for each course iteration is crowdsourcing. The specific uses of crowdsourcing in computer science education are discussed further in the next subsection.

2.1.2 Systems in computer science education

Computer science education is a field of science that concentrates on teaching and learning computing, computer science, and computational thinking [39]. Besides physical textbooks, the field heavily utilizes online textbooks and different tools to both improve the learning experience and collect research data from the student populations. These tools can, for example, try to teach programming syntax [40], ask students to arrange code blocks into a correct order [41], or require students to create their own exercises [6, 10, 11, 12]. This thesis concentrates on the last-

[7]https://www.coursera.org/

mentioned tools, as they can be used to crowdsource content while also working as practice and revision tools for students.

The main benefit of using crowdsourcing in computer science education is balancing the workload required for creating new exercises and updating course materials. There are two main ways of using crowdsourcing to create materials – using a crowd of students, or using a crowd of educators.

Using students as a crowd is usually preferred, especially if the materials to be produced are simple, but the amount required is great. Generally, introductory programming courses attract more students than the advanced courses, offering a large group of students for crowdsourcing activities with each iteration. Crowdsourcing activities can be merged with the practice portion of the course so that the activity itself is more or less hidden from the student – instead, they focus on completing an exercise, or revision. Thus, learning and collecting happen simultaneously, in preferable cases with minimal overhead for the student. The most notable disadvantage of using students as the crowd comes from the unbalanced skillsets and varying levels of expertise. Since most students, especially on introductory programming courses, are novices in the topics they are studying, it is natural to doubt the capabilities of the crowd when it comes to creating viable materials via crowdsourcing. The effects of a mostly novice crowd are investigated in this thesis – however, based on previous studies [7, 8, 9], the end product quality is not estimated to be drastically different compared to using more experienced crowd, as long as the crowdsourcing tasks are presented in a meaningful way.

Using a group of educators as a crowd has its benefits compared to using students. The quality concerns mentioned previously should not rise as an issue. Besides this, teachers and instructors have a concrete understanding of the curriculum and the course contents, as well as the topic at hand. Thus, they have a better grasp of what kind of material is useful and necessary for the learning process, meaning that the way crowdsourcing tasks are presented need not be as strictly stuctured, unless a very specific type of material is required. Educators also understand the different levels of comprehension a student may have on the topic, and are more capable of creating exercises of various difficulty levels. Moreover, especially face-to-face crowdsourcing sessions may be beneficial in networking, and can encourage educators to seek collaboration in other ways as well. However, as one of the main reasons to use crowdsourcing in educational purposes is alleviating the workload educators have, this method does not provide clear advantages over the regular situation. Teachers and instructors may have restrictions on their available time and resources, and participating in crowdsourcing of materials rarely substitutes parts of existing work during the creation process, thus adding to the workload instead of easing it. Some instructors may also feel that they are not gaining enough from the end product for it to necessitate their input. Sharing previously created materials for common use might feel unfair if the gain is not clear enough.

The rest of this subsection presents computer science education tools and material banks that utilize crowdsourcing in one way or another. The first three tools, Peer-

Wise, CodeWrite and StudySieve, are used by students, who also act as the crowd for the crowdsourcing efforts. These tools are similar to CrowdSorcerer, which is introduced in detail in Section 3.1. The last three are used by educators, and present a wider variety of crowdsourcing tasks. Besides introducing the relevant systems and material banks, this subsection also discusses some general benefits and difficulties these systems may have.

PeerWise [10] (Figure 3 from the documentation[8]) is a web-based tool used to crowdsource multiple choice questions. While the format of multiple choice questions makes them applicable for wide range of educational fields, this background literature review focuses on the studies on the usage of the tool in computer science courses.

When creating a multiple choice question, the user provides a question stem and two to five answer alternatives, as well as the indication of the correct answer, and an explanation of why the marked answer is correct. All of the questions entered to the system are available as exercises for other users. After completing the question, the answering user can give feedback and rate the exercise. The user who created the question can track how their exercise is doing, whether it is rated overtly difficult or not, and if there are any mistakes in the question stem or the answer.

PeerWise is largely a self-correcting system, as mistakes in short multiple choice questions are easily spotted by those who try to solve the exercises. This was thoroughly investigated by Denny et al. [42], and it was noted that even if the exercises originally had some mistakes, all of them were corrected through feedback. The system has also received very positive feedback from students who have described the system as a very good way to review concepts of the course [10, 43]. The ever-growing number of multiple choice questions also benefits the course instructors, as they have practice material available even at the beginning of a course, collected from the previous iterations.

CodeWrite [11] (Figure 4) is a web-based programming education tool that provides drill and practice support for Java programming. The user first writes both a brief and a more detailed description of the purpose of the method they need to construct. The user also creates the descriptions and other necessary information needed to complete the exercise, such as the return type and the method name, as well as parameters of the method. After filling out the method as instructed, the user creates test cases for their program. These test cases are used directly to test that the implementation given works as intended. The user receives direct feedback of their exercise through the test results. All tests must pass before the exercise is shared with other users. When published, other users can attempt to solve the exercises that have been created by peers on the course.

Based on student reports, CodeWrite helped the users to learn programming concepts, as they spent time investigating each other's solutions and comparing them to their own [11]. The same study also noted that the exercise database built using only student-generated exercises covered all the topics on the course, giving users a

[8]https://peerwise.cs.auckland.ac.nz/docs/students/

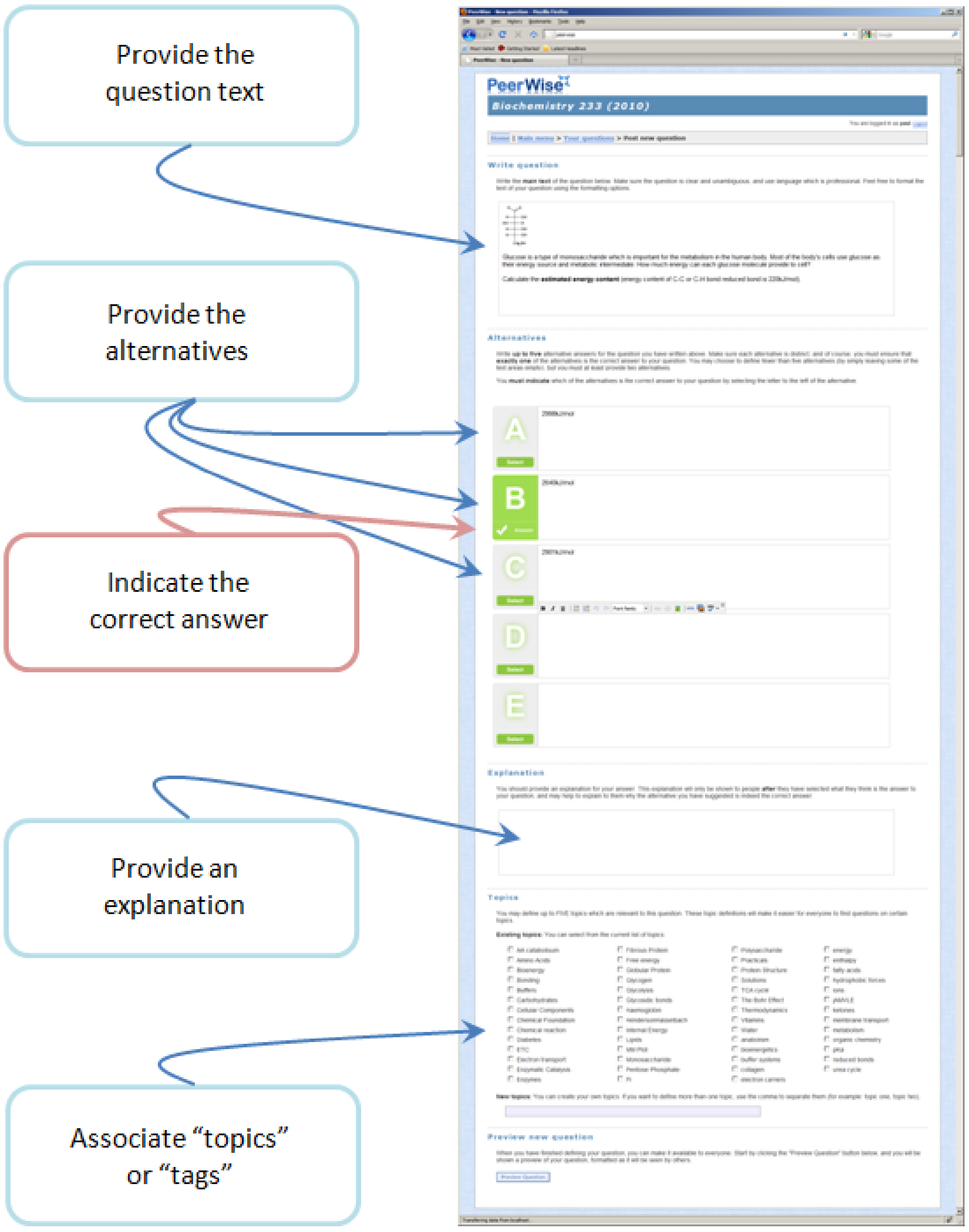
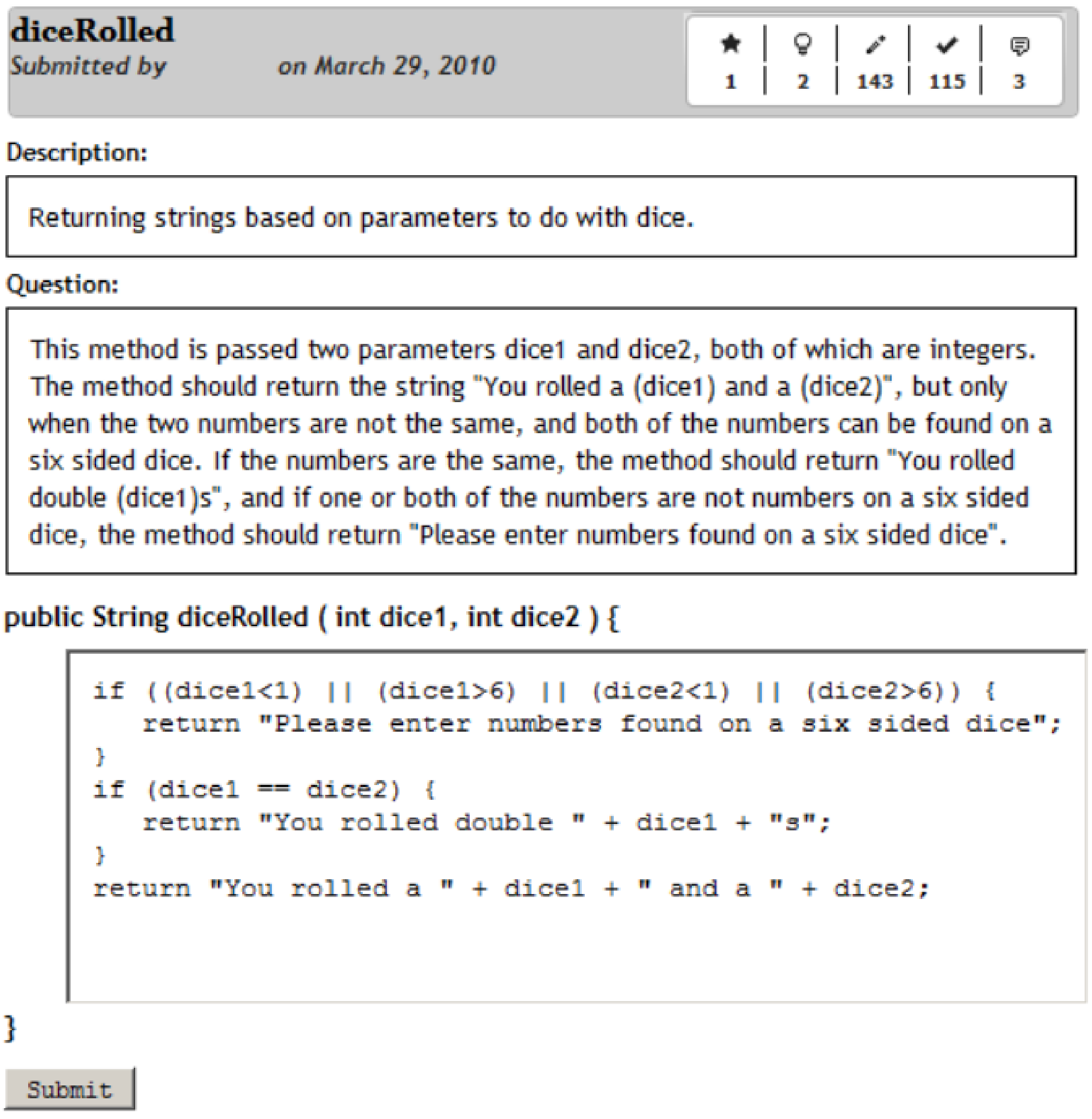


Figure 3: Overview of the question creation process in PeerWise [11].



Description:

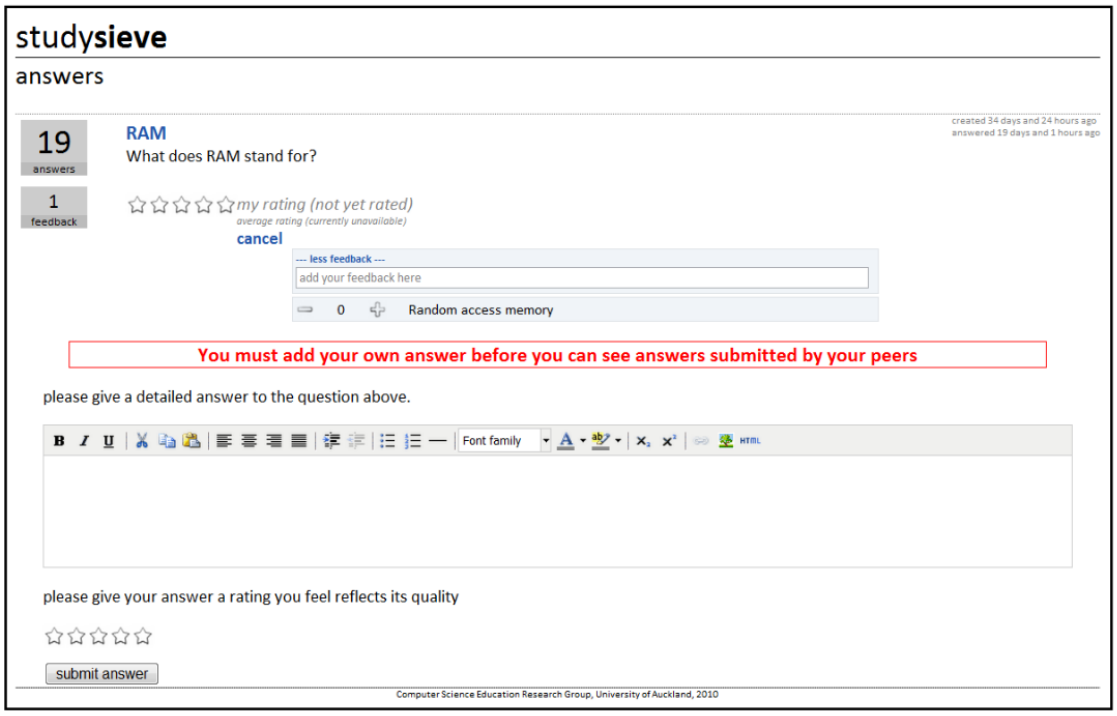
Returning strings based on parameters to do with dice.

Question:

This method is passed two parameters dice1 and dice2, both of which are integers. The method should return the string "You rolled a (dice1) and a (dice2)", but only when the two numbers are not the same, and both of the numbers can be found on a six sided dice. If the numbers are the same, the method should return "You rolled double (dice1)s", and if one or both of the numbers are not numbers on a six sided dice, the method should return "Please enter numbers found on a six sided dice".

```
public String diceRolled ( int dice1, int dice2 ) {
    if ((dice1<1) || (dice1>6) || (dice2<1) || (dice2>6)) {
       return "Please enter numbers found on a six sided dice";
    }
    if (dice1 == dice2) {
       return "You rolled double " + dice1 + "s";
    }
    return "You rolled a " + dice1 + " and a " + dice2;
}
```

Figure 4: CodeWrite [11]. The student enters their solution to the question into the provided text field.


studysieve
answers

19 answers
1 feedback

RAM
What does RAM stand for?

created 34 days and 24 hours ago
answered 19 days and 1 hours ago

my rating (not yet rated)
average rating (currently unavailable)
cancel

--- less feedback ---
add your feedback here
0 Random access memory

You must add your own answer before you can see answers submitted by your peers

please give a detailed answer to the question above.

please give your answer a rating you feel reflects its quality

submit answer

Computer Science Education Research Group, University of Auckland, 2010

Figure 5: StudySieve [12]. Students must complete and rate their own answer before they can see the answers of the other students.

comprehensive set of exercises to practice on.

Since exercises created on CodeWrite are published for use as soon as they pass the tests, there have been some issues with the exercise description and demanded method not matching [44]. This causes issues for those attempting to solve the exercise, as they will not get the model solution before they submit a working solution of their own, which can lead into a guessing game if the description is vague or erroneous.

StudySieve [12] (Figure 5) is a web-based tool for forming free-response questions. Similarly to PeerWise, while it is possible to use StudySieve on various fields besides computer science education, this background literature review focuses on the usage of the tool on undergraduate computer science courses.

The user creates a question according to the guidelines given by the instructor. The guidelines are to remind of checking the quality of the question before submitting it, as well as to remind of the preview function to make sure the submitted question is formatted properly. To avoid repeated questions, the question text the user is writing is compared to the database of existing questions, and similar ones are shown below the text field. This encourages the user to alter their own question, and decreases the number of duplicates.

The existing questions can be answered, and though there are no checks for the correctness of the solution, the answering user only sees the other possible solutions to the question after they have submitted their own answer. Then, they can compare their answer to others', and use the peer reviewing functionality of the system to discuss the answers submitted, as well as the question itself. The peer reviewing functionality for both the questions and the answers includes statements that are rated from one to five (such as "Correctness of the answer") and a field for written feedback.

Studies on StudySieve state that the student responses to the use of the tool were very positive, and that the students enjoyed creating more questions than it was required, as they believed it helped them to revise and learn concepts of the course [45]. It has also been found out that those students who actively participate in these types of question-and-answer creation activities perform better in exams, even when considering their prior abilities [46, 47].

As mentioned before, crowdsourced materials created by educators can be very different from those created by students. Materials crowdsourced from students are usually simple exercises with their answers. To get the most out of the educators' time and resources put into the crowdsourcing efforts, the materials produced are usually more on the line of course content, such as online text books, visualisations, or extensive sets of exercises.

Canterbury QuestionBank [48] is a comprehensive set of multiple choice questions suitable for first-year computer science courses, developed at ITiCSE 2013 conference by a working group of computer science educators. The working group produced over 650 questions and answers suitable for, for example, quizzes and exams. The question bank can be found and contributed to through its website[9].

Each of the exercises consists of the question itself, the correct answer with an explanation, at least one other answer alternative, tags to describe the question and its topic, and quality and difficulty ratings, as well as some additional comments from reviewers. All the questions in the question bank are created and reviewed by instructors to ensure good quality and beneficial assignments.

The working group identified twelve distinct patterns for multiple choice questions [48]. These include, for example, fixed-code debugging (finding an error from the code given in the question), purpose (given a code segment, explain the purpose of the particular piece of code), and algorithm and data structure tracing (such as order of nodes visited in in-order traversal).

CodingBat [49] is an online repository[10] of small programming exercises in Java and Python. The idea of the site is to give the user lots of short exercises with simple descriptions and immediate feedback. While coordinated by a computer science lecturer at Stanford University, exercises can be added to the repository by teacher profiles. Since CodingBat is not crowdsourced through users, but through instructors, the repository is noticeably smaller than those using much larger masses

[9]http://web-cat.org/questionbank/
[10]https://codingbat.com/

to build up the database.

OpenDSA [50] is a collection of online materials meant to support a wide range of computer science courses, such as data structures and algorithms, programming languages, and formal languages. The materials provided by the system are available on the webpage[11] of the project. The materials include visualisations, full course books, exercises and singular modules that can be used as a part of any course material.

Contributing to the OpenDSA collection happens through its GitHub repository's[12] pull request feature, which makes it possible for the original developers to review the additions the contributor wishes to make. The pull request feature enables line-by-line commenting, which makes it easy to point out issues in the contribution, and opens a place for discussion.

Results of learning from student-generated content have been very promising as well [46, 47, 51, 52, 53, 54]. Again, when examining the use of StudySieve, Luxton-Reilly et al. [53] found a positive correlation between question-generation exercises and exam performance that cannot be simply explained with prior experience or knowledge, though the exact effect of the tool usage is very difficult to measure due to variables like the pressure of the exam situation. Revision of topics for exams has been mentioned as a clear benefit for PeerWise and CodeWrite, while systems like CodingBat can act as a supporting revision measure without active crowdsourcing from the students. For example Denny et al. [43] report that students create, answer and review more multiple choice questions than is required of them, especially right before final exam, and student reports say that the tool is very practical for intensive revision and recalling the most important topics at the end of the course.

The use of PeerWise has also been studied to encourage deep learning as opposed to surface learning, as the students need to analyze the presented exercises on a deeper level for the reviews [10]. Using the database for purely revision without the creation and reviewing parts would reduce the system to a simple drill practice. Deep learning is also supported through reflection, as in order to create relevant multiple choice questions, the students must first understand and reflect on the learning outcomes of the course [10]. The students must also be able to give alternative answers besides the correct one as distractors, which means that they need to know their question topic well enough to come up with possible misconceptions that might occur. Sufficient understanding of the topic at hand is also required to properly reason why one answer is correct and why the other ones are not.

Denny et al. [10] note on their analysis of PeerWise that one significant benefit of crowdsourcing systems outside of the range of student learning experiences is the time saved by collecting a large question bank through the use of the system on courses. Besides getting a repository of potential quiz exercises, the database can also be used for analysis of student performance and common misconceptions. The same benefits can be applied to StudySieve and CrowdSorcerer, and to some extent,

[11]https://opendsa-server.cs.vt.edu/
[12]https://github.com/OpenDSA

CodeWrite, though the latter lacks the increased quality of reviewed exercises. From another point of view, systems like Canterbury QuestionBank and OpenDSA provide instructors ready-made materials to use on their courses, which also serves as a time-saver.

Denny et al. [8] also examine the coverage of course topics in a database generated with PeerWise, and conclude that the database covers all the major topics of the course, despite students having complete freedom on choosing the topics for their questions. The same study reports that a large part of the questions used multiple topics from the course, sometimes creating very complex exercises, which indicates that the problem of students taking the easy route and creating only very simple exercises to spend as little time as possible with the tool was not observed.

Weld et al. [55] note the difficulty of personalisation on online courses, and propose to solve some of the presented problems with crowdsourcing. Their solutions include, for example, peer reviewing processes for support through feedback and engagement through a large database of crowdsourced exercises that can be automatically given to students depending on their process on the course. This is directly what tools like PeerWise, CodeWrite, StudySieve and CrowdSorcerer strive to achieve.

The most pressing difficulty encountered with crowdsourced materials is with the quality of the produced content. Every person contributing is not as talented or well-versed in the field they are partaking in. For example, when creating programming assignments, novice programmers may not yet fully understand what is required from a good description, or they may have a lacking knowledge of good testing practices, such as test coverage. This might allow mistakes to slip into the finalized exercises, even if the systems have some checking measures, like for example CodeWrite and CrowdSorcerer have. On the other hand, experienced programmers might find it hard to create an assignment that is simple enough for the level of the introductory programming course they need to participate in, as many programming practices not yet addressed on the course may seem self-explanatory to them.

One way to combat the quality issues is expert reviewing, or a system that only accepts contributions from experts. CodingBat is one example of a system using the latter approach. While the quality of the exercises in the system is guaranteed to be good, it is very clear that the system suffers from the lack of exercises when compared to, for example, the larger databases constructed with tools like CodeWrite. Fewer exercises means that students will go through them more quickly, have less materials to revise from, and will require exercises from other sources, which in turn takes up the instructor's time. The time and resource limitations are also a reason why expert reviewing does not seem to be a popular choice for crowdsourcing quality management in computer science education. Joint projects like Canterbury QuestionBank can produce a large database through one coordinated working group effort, but also suffer from lack of updates – only two questions have been added since the publication of the original article in 2013.

Another way to avoid quality issues is to use peer reviewing. This procedure can be seen in use in PeerWise, StudySieve and CrowdSorcerer. While it would seem that it is risky to use the same students who may produce wildly erroneous assignments to review said assignments, multiple studies [8, 9] show that the quality of peer reviewed assignments can be good. There are no significant differences in peer reviews and expert reviews [9], nor with reviews done by novices or more experienced programmers [7, 9]. That being said, a study by Hui et al. [56] notes that anonymous reviews are more honest than those with the reviewers name attached to it, so the way the review process is handled may affect the end results noticeably.

Some systems, such as CodeWrite, work without any review measures. However, Denny et al. [44] do note that student feedback has indicated problems such as vague exercise descriptions, mistakes in program code and so forth. This presents an obvious quality issue, and causes issues for both the students and the instructor. The students may get misleading information from erroneous exercises, or may not be able to complete some exercises due to lacking descriptions. On the other hand, the instructor can not use the exercises as a database for following iterations of the course, at least not without running a laborious review process beforehand.

Besides quality issues, one difficulty the crowdsourcing process itself might run into is lack of crowd interest. For example, only third of the course population tried CrowdSorcerer in its first iteration [6]. Lack of data leaves the database small, and makes the process of collecting exercises slower. Similarly, small set of data can hinder studies of the tool, as the userbase is too marginal to present any relevant results. The same study notes that students were more keen on peer reviewing than creating assignments, which might be due to seemingly larger workload of constructing an assignment from scratch.

Preparing course materials takes time, and though many instructors want to make their courses look their own, it is still very useful and time-saving to have ready-made, freely usable databases of course materials online. The labour needed for creating course materials discourages creators from openly sharing their content, which means that many potential tools stay in their university indefinitely [57]. Good quality open materials like OpenDSA are difficult to come by, and require extraneous effort from at least instructor-level expert to manage.

When it comes to the benefits of crowdsourcing in computer science education, there is a multitude of great experience reports available. The current research implies that crowdsourcing is a very usable and beneficial method when it comes to educational purposes, and that the difficulties it can bring are mostly avoidable, as discussed above. The benefits can be seen more clearly with extended research, as has been noted for example with PeerWise and CodeWrite. Continuous research in varying environments, such as different universities or subjects, gives all the more reliable overall picture of the effects of these tools and materials.

2.2 Quality

One key term in this thesis, also present in the title, is quality. In the context of software development, quality is a concept that is somewhat hard to define. Usually, what is seen as a good quality product or process depends on the phase the development is in, such as production or integration [58, 59]. Thus, quality can concern the security of the software, whether it fulfills the goals set for it, how readable the code is, how detailed the documentation is and so forth. In the context of this thesis, quality under evaluation is defined as the suitability for a specified task, that is, how well the assignments produced with CrowdSorcerer meet their goal and whether this can be measured.

Garvin [60] uses five different views to inspect the prospect of quality in various fields such as philosophy, economics and marketing. These five viewpoints have been applied to various other fields [61, 62] beyond Garvin's original description, including software quality management [59, 63]. The most relevant for this thesis are the following three views.

- *User view* sees quality as the fit of a product to its intended purpose. It expects that the quality of a product or idea is supposed to bring some kind of benefit to its user, and the better the product or idea benefits the user, the greater quality it also is. User view of quality can be broadened to inspect wider userbases, or to the level of an individual – how well the product or idea serves its entire intended audience as a whole, or how beneficial and usable it is for each singular user.
- *Manufacturing view* inspects quality from the perspective of conformance to specifications. It is assumed that there is a plan or a specification document that is used at the producing phase, and the quality of the item is evaluated only through how closely the finished product resembles the specification. This can be a hindrance from the user view. If the specification is faulty, the end product will also contain these errors, if manufacturing view is the sole measurement for quality.
- *Product view* evaluates quality as something tied to the inherent characteristics of the product. In the case of software quality, this would mean valuing the inherent quality of the program more than the external, such as focusing on the documentation and readability of the code, and expecting that internal quality will inherently produce external quality (user view quality) as well.

As mentioned before, Garvin's views for quality can be used to assess quality in various fields. Kitchenham and Pfleeger [59] focus on describing measuring the quality of a software product from user view and manufacturing view. Measuring quality requires not only the definition for quality, but also baselines. For example, one mentioned baseline for measuring quality from user view is reliability, that is, how long the product is able to function without failures [59, 64]. Other possible measurement of quality from the user view is usability, including everything from installation of the software to learning to use it, and performing the tasks with the software regularly. Both examples here, reliability and usability, are possible to be measured using set characteristics, like time between failures for reliability, and time spent on learning the use of software for usability [64, 65].

Similarly, Kitchenham and Pfleeger [59] give two characteristics to measure for the manufacturing view when it comes to software quality: defect counts and rework costs. Defects are recorded during the development process and usage across all modules, and must be evaluated in the same way throughout the entire process. Rework costs relate closely to defects – while some defects are very minor, and easy to fix when found, while some are disastrous to the entire system. The latter defects can end up being very costly, causing issues before they are found and being tricky and laborious to fix when encountered. This increases the rework cost, that is, the effort required to report and fix defects during production and after launch.

Tian's article [66] presents evaluation models as a way of estimating and measuring software quality. The article divides software quality characteristics roughly into two categories: those that are directly related to functional correctness or to the conformance of specifications, and those more on the user-side of the product, such as usability and portability [67, 68]. Evaluation models can be used to identify defects, error-prone areas and predict resulting quality, thus improving project management, optimisation and resource handling [69, 70].

There are similarities between Tian's models [66] and the quality viewpoint for software engineering presented by Kitchenham and Pfleeger [59] from Garvin's quality viewpoints [60]. Both concentrate heavily on defects, Kitchenham and Pfleeger giving special attention to the effects such as reworking costs in the description of manufacturing view of software quality. Kithcenham and Pfleeger also concentrate more on usability through the user-side view, while Tian's perspective is heavily on the manufacturing view. Kitchenham and Pfleeger actually mention quality models in their article, though they introduce a wider variety of models than Tian. Both articles mention ISO 9126 [68], perhaps the most well-known code quality standard, that defines quality through six characteristics: functionality, reliability, usability, efficiency, maintainability and portability. As many of these characteristics are heavily from the user view of quality, Tian does not put much emphasis on these.

Conversely, Dromey [71] states that the term software quality has been used too loosely in relation to the product and the process. He argues that this has created confusion and diversion from the primary goal of the industry – that is, improving the quality of the software during various phases of software development. Dromey introduces a quality model framework to alleviate the semantic confusion and vague terminology of quality to distance the conversation from subjective ideas of "goodness" and "fitting the purpose". He also states that the idea "quality should be built into software" inherently distracts from the real issue of how to build a software that has high-quality characteristics.

Dromey notes that he is not the first one trying to tackle the problem of software product quality in a systematic and comprehensive manner [72, 73, 74]. However, previous attempts of building a suitable quality model framework have stalled, for example, because of the diversity of defects in software, and the perceived scale of the quality problem in general. To overcome these hindrances, Dromey proposes a strategy of always proceeding from the tangible and measurable quality characteristics to the less tangible, higher level ones. To accomplish this, Dromey presents a generic quality model and a process to build such a model for different software products. The details of this model are outside of the scope of this thesis – see [71] for details.

2.2.1 In computer science education

In general, the concepts of quality that software engineering uses also apply for computer science education. This section mainly focuses on inspecting and improving students' code quality. As novice programmers may not write quality code by themselves, the job of the instructors is to ensure that not only are the examples in the course materials of good quality, as this is what students mainly use as reference when they write their first programs, but also that students learn to pay attention to what features good quality code has.

Measuring students' code quality is not a simple task, as it cannot be assumed that novice programmers can adhere to the quality standards commonly used in the working life [68]. Thus, code quality measurement studies tend to end up rather context specific [75]. In their study, Breuker et al. [75] compared first year computer science students' code to that of second year students, and found no significant differences in the static code quality with their measurements. They do note, however, that their results may be skewed due to the more complex programming assignments the second year students are expected to complete, and how their measurement system does not suitably acknowledge this change, thus rating programs it deems too long or complex as low quality.

One gentle way of introducing novice programmers to code quality management is through code reviewing, which can be approached through pair programming. Studies have found many benefits for pair programming in general, including increased confidence and performance, improving laboratory experience, and fewer "give-ups" due to lack of teaching assistant's time [76, 77, 78]. During pair programming, students naturally participate in code review, even if it is not systematic, as they continuously read through each other's code and correct errors as they proceed.

Guided peer code reviews and peer assessment has been used in introductory programming courses with promising results. For example, a study by Brown [79] found out that their guided peer code review system helped students to understand hard-to-comprehend programming concepts, improved their programming skills, and made them better at finding errors in programs. The same study also notes that improved performance during the code reviews translates to the improvement in the quality of the code produced. Similarly, Hamer et al. [9] report good correlations in their study for quantitative marks given by students and tutors, with improving correlations as student ability and experience increase.

Stegeman et al. [80] report their experiences using a code quality assessment rubric meant to provide feedback from instructor to student. Based on their previous work [81] with assessment of code quality in introductory programming courses, the studies disassemble existing models of code quality and structure the introductory-course-relevant aspects into concrete feedback tools. Working through multiple iterations of the rubric, Stegeman et al. present a working feedback tool for systematically assessing code quality aspects like naming, formatting, and flow of the code. For clarity and learning purposes, the rubric includes descriptors for the achievement levels and criterion, so that the rubric is not only a helpful grading tool for the instructors, but also an instructional document for the students.

One other possible perspective into code quality management comes in the form of test-driven development (TDD) [82]. In TDD, the programmer specifies the test before implementing a new portion of the program. Typically, testing and especially TDD practices are not introduced very early during introductory computer science studies, as it is not seen as a framework that is easy for novice programmers to grasp effectively [83, 84]. Mugridge [84] notes that not only does TDD require skills in testing, design and refactoring to utilize it well, the students also require a complete reconfiguration of their models of learning and designing programs. Thus, one important part of teaching both testing and TDD is to help students understand and appreciate testing, and not see it as a redundant hurdle in completing a programming assignment [85]. Some teachers also feel that they barely have enough weeks on a course to teach the required topics as is, and adding testing would only increase the overflowing workload [86].

While the approach is not the most common one, many courses have tried including TDD into curriculum very early in the computer science studies. For example, Desai et al. [86] report on their experiences in integrating TDD into existing CS1 and CS2 materials without reducing topic coverage or increasing instructor workload. Through exposing students to testing through examples, and gradually increasing the number of tests required per programming project, students were smoothly guided into writing tests unprompted and on their own. The same results have also been previously noted by Janzen and Saiedian [87]. Desai et al. [86] also noted that while awarding points for test code did not significantly change the quality of the source code, attitude towards testing, or comprehension of the material, it did give students more incentive to create more test cases, thus producing higher quality code measured through code coverage. Edwards [83] reports very similarly, stating integrating TDD into the introductory programming course helps students distance themselves from the trial-and-error approach of debugging and encourages them to learn analytical testing skills.

2.3 Inter-rater reliability

Inter-rater reliability is a statistical method used to describe the degree of agreement between two or more raters or reviewers. Different statistics of inter-rater reliability can be used to calculate the score of consensus between ratings of different reviewers (in contrast to *intra-rater reliability*, where the consistency of ratings for one judge in multiple occasions is measured).

Inter-rater reliability can be used to, for example, refine tools that are given to human judges by determining a scale that is appropriate for measuring the particular variable in question. Low inter-rater reliability means that either the scale used is defective, the rating instructions are unclear, or the raters need to be retrained.

There are multiple statistical methods for inter-rater reliability measurements in different situations and for different data sets, such as Cohen's kappa (for two raters) [88], Scott's pi (for nominal data) [89], and Fleiss' kappa (for fixed number of raters and categorical data) [90]. The statistic used in this thesis is Krippendorff's alpha for its suitability for various types of data.

Krippendorff's alpha [91] has been used since 1970s in content analysis as a general agreement measure with appropriate reliability interpretations (for example [92, 93, 94]). Krippendorff describes that the α-agreement gives uniform reliability standards over a wide array of varying types of data [91]:

- It is applicable to any number of values per variable. Its correction for chance makes α independent of this number.
- It is applicable to any number of observers, not just the traditional two.
- It is applicable to small and large sample sizes. It corrects itself for varying amounts of reliability data.
- It is applicable to several metrics (scales of measurements) – nominal, ordinal, interval, ratio, and more.
- It is applicable to data with missing values in which some observers do not attend to all recording units.

In its most general form, the definition of α is

$$\alpha = 1 - \frac{D_0}{D_e},$$

where D_0 is a measure of the observed disagreement and D_e is a measure of the disagreement that can be expected in case of a chance [91].

The above definition has two implications relevant for reliability measurement. First, when the observed agreement is perfect and therefore, the disagreement is absent, $D_0 = 0$ and $\alpha = 1$, which indicates perfect reliability. Coincidentally, when agreement and disagreement are random, and observed and expected disagreements are equal, $D_e = D_0$, $\alpha = 0$, indicating the absence of reliability [91].

The α value could also become negative, as much as -1 [91]. However, negative values are considered a result of two kinds of errors: sampling errors and systematic disagreements. As the aim is to achieve as high reliability as possible, negative values are too divergent from the reasonable reliability values to matter. For reliability considerations, the values of α are limited to

$$1 \geq \alpha \geq 0,$$

$\pm$ sampling error and $-$ systematic disagreement [91].

The more detailed definition of Krippendorff's alpha beyond this description is not within the scope of this thesis, but can be found in *Content Analysis: An Introduction to Its Methodology* by Klaus Krippendorff [91].

Sampling errors can be a result of multiple reasons, for example, too small sample sizes. With only a few observations, each rating has a large effect on α. When the observed disagreements do not line up with the expected disagreements, the α values fluctuate above and below zero. Systematic disagreements occur when raters have different interpretations for the instructions given to them, or simply disagree heavily on the topic. While all observed disagreements waver the perfect reliability, systematic disagreements can cause α values to drop below what could be expected by chance [91].

The minimum acceptable value of the α coefficient should be chosen according to the importance of the conclusions that are to be drawn from the data, even in the case the data is imperfect for example due to sampling errors [91]. The higher the cost of erroneous conclusions, the higher the minimum alpha should be. When the risks of drawing false conclusions from unreliable data is not known, social scientists rely only on data with reliability of $\alpha \geq 0.800$. Data with reliability of $0.800 > \alpha \geq 0.667$ is considered viable for drawing tentative conclusions, and data with agreement of $\alpha < 0.667$ is not taken into account. For conclusions that allow more lenient measurements, Landis and Koch [95] have set magnitude guidelines as follows:

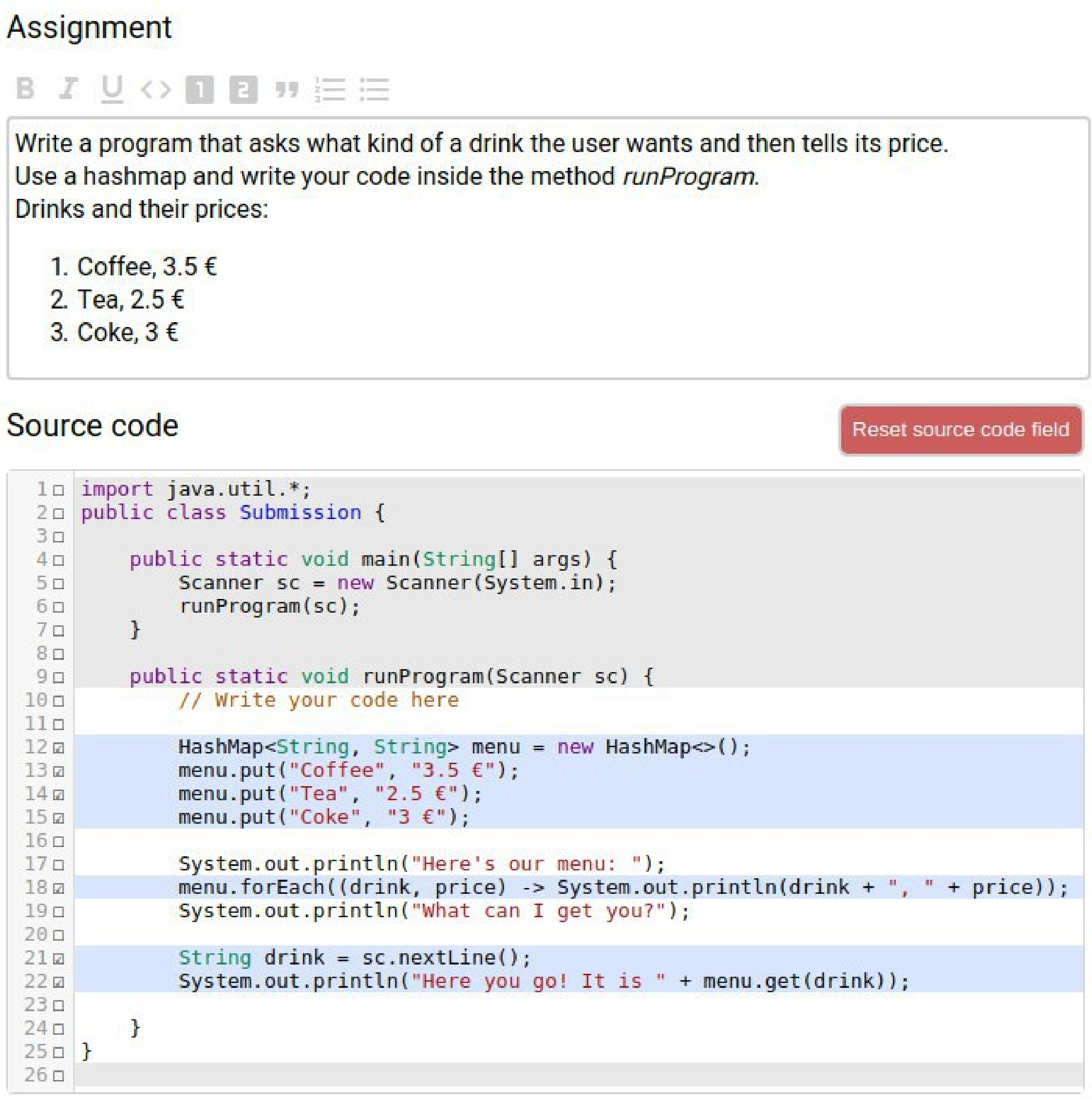
- 0.81-1: Almost perfect agreement
- 0.61-0.80: Substantial agreement
- 0.41-0.60: Moderate agreement
- 0.21-0.40: Fair agreement
- 0-0.20: Slight agreement
- Less than 0: Agreement random or worse

The latter guidelines by Landis and Koch are used for analysis of the results of this thesis in Sections 4 and 5.

3 Methodology and research design

The following subsections go through the methodology, data collection, and research design of this thesis. Section 3.1 gives the system description of CrowdSorcerer, the system used to crowdsource programming assignments in this study. Section 3.2 describes the data collected with CrowdSorcerer. Section 3.3 introduces the five research questions, and Section 3.4 the expert review criteria.

3.1 System description



Write a program that asks what kind of a drink the user wants and then tells its price.
Use a hashmap and write your code inside the method *runProgram*.
Drinks and their prices:

1. Coffee, 3.5 €
2. Tea, 2.5 €
3. Coke, 3 €

```
import java.util.*;
public class Submission {

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

    public static void runProgram(Scanner sc) {
        // Write your code here

        HashMap<String, String> menu = new HashMap<>();
        menu.put("Coffee", "3.5 €");
        menu.put("Tea", "2.5 €");
        menu.put("Coke", "3 €");

        System.out.println("Here's our menu: ");
        menu.forEach((drink, price) -> System.out.println(drink + ", " + price));
        System.out.println("What can I get you?");

        String drink = sc.nextLine();
        System.out.println("Here you go! It is " + menu.get(drink));

    }
}
```

Figure 6: The basic assignment creation view of CrowdSorcerer. At the very top, the assignment field contains a student-written assignment handout. The source code field has a student-written code with the model solution marked with the checkboxes on the left (blue lines). The lines in gray are used as a base for the program, and they cannot be edited by students.

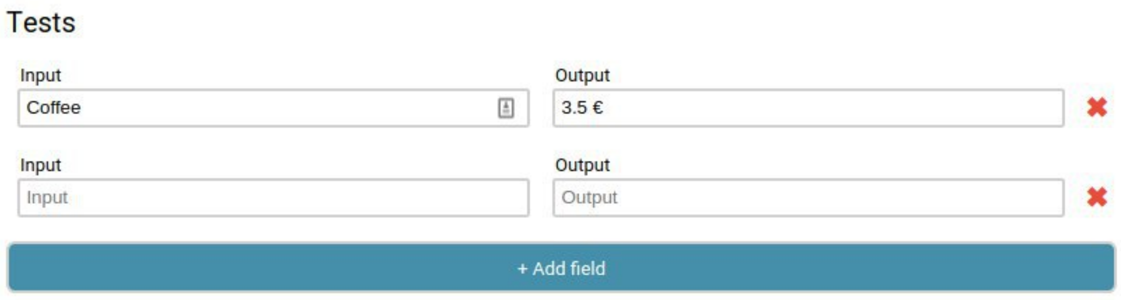


Figure 7: Simple input-output test cases.

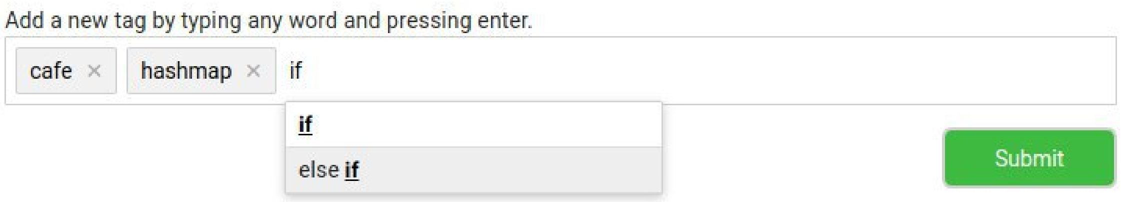


Figure 8: Tagging system with suggestions to make categorizing assignments by topics easier.

CrowdSorcerer [6] is an embeddable tool that is used for programming assignment creation. Development of the tool started in the summer of 2017 by the author and her colleagues in the Agile Education Research Group (RAGE) at the University of Helsinki. The frontend[13], that is, the embeddable user interface of CrowdSorcerer, is built with React. The backend[14], which holds most of the system's functionality and storage, is a Ruby on Rails application. The student-created assignments are sent to a Test My Code programming assignment evaluator server [96][15].

The user interface for basic assignment creation can be seen in Figure 6. First, the user comes up with an assignment handout according to specifics given by the instructor. These instructions may require, for example, usage of conditional statements. Then, the user programs a model solution (seen in blue lines in Figure 6) and code template for the assignment they came up with (everything else in the source code). The model solution is a full, working answer to the assignment, while the code template only contains the basic structure of the program without the crucial implementation lines. In order to help the student to recognize the relevant lines of code for the code template, and to keep the source code functional, parts of the code can be locked by the instructor so that the students are not able to edit

[13]https://github.com/rage/crowdsorcerer
[14]https://github.com/rage/crowdsorceress
[15]https://github.com/testmycode

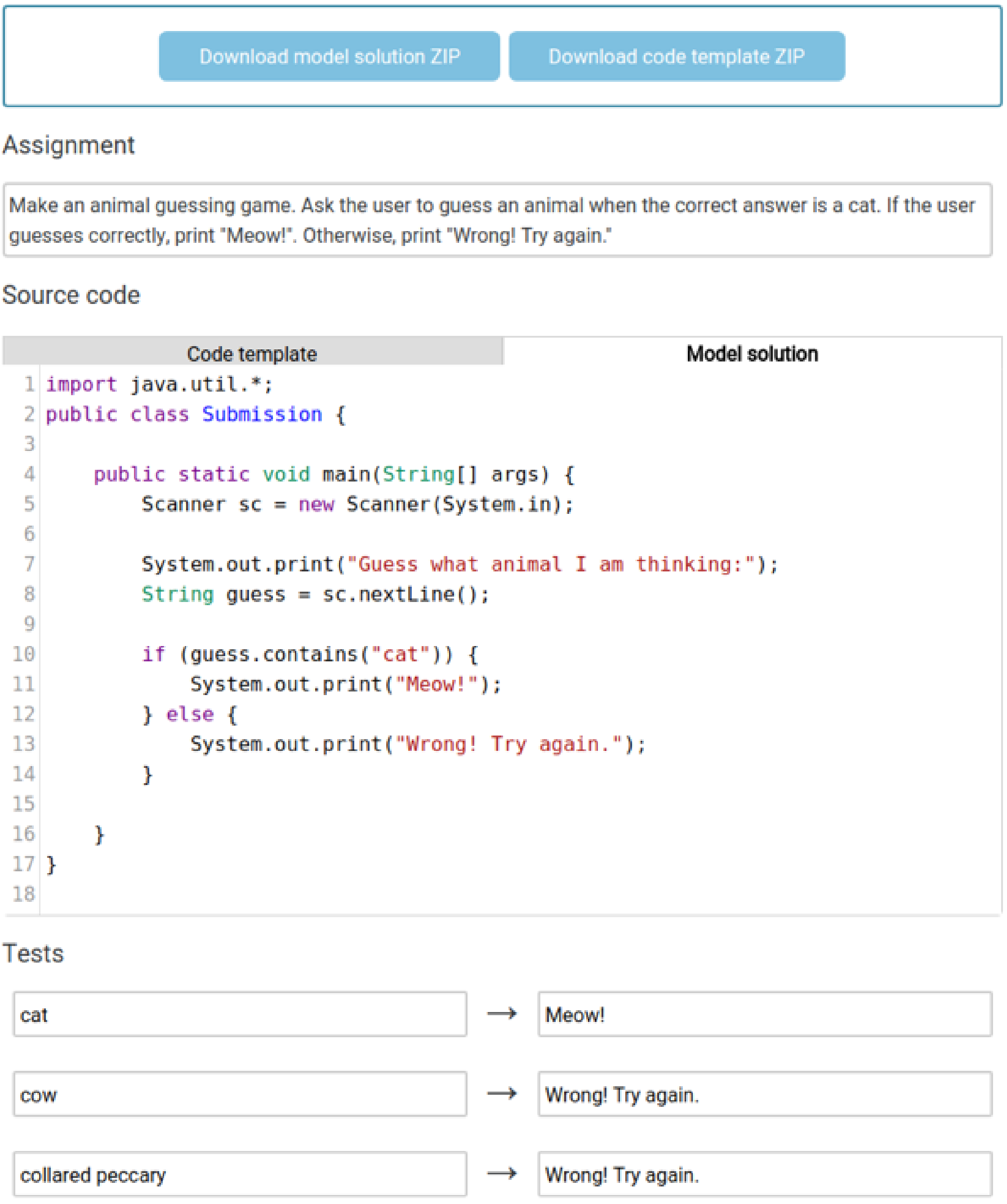
these lines (seen in gray lines in Figure 6). The user also invents some test cases for their program (Figure 7). The program code is automatically tested for compilation errors, and the user-given tests make sure that the program works as expected.

The created assignments can then be peer reviewed. The reviewer can inspect both the source code and the model solution, as well as the given tests (Figure 9). They are required to answer review statements chosen by the instructor, such as "Exercise is suitably difficult" and "Test inputs and outputs are reasonable" (Figure 10). The reviewer is also prompted to give written feedback. The number and content of review statements are decided by the instructor while setting up the CrowdSorcerer instance. The default minimum for the written feedback is three words.

The instructor can set the number of assignments to review for each instance of CrowdSorcerer. The default is three assignments for each student. If the student created an assignment, they will receive their own assignment to review as the last shown assignment. This is to encourage them to compare and reflect their work to that of their peers'. If the student did not create an assignment for the instance, they will receive the set number of other students' assignments to review. It should be noted that while the students need to be logged in to the course material in order to create and review assignments, the system handles all the processes anonymously from the students' perspective.

Both the creator and the reviewer are also required to give the assignment at least one relevant tag. Tags are supposed to describe the assignment in a very brief manner – descriptive tags can be, for example, *for-loop*, *if-else*, *conditionals*, *easy* and so forth. The tool gives tag suggestions from its database, as can be seen in Figure 8. Tagging helps with identifying the assignments by their features, for example their topic or difficulty level. Cross-referencing the tags from both the creator of the assignment and the reviewers increases the correctness of the tagging system.

Many instructor-specific features of the tool can be used through a separate administrative user interface (UI), accessible through a browser. Through the admin UI, the instructor or course assistant can create new assignment instructions, inspect the created assignments with full information and relevant peer reviews, set the tags the tool suggests and adjust the peer review statements. If need be, assignments and users can be deleted, so reacting to any reports of misbehavior is easy. The admin UI also makes it convenient to inspect the statistics of each instance of CrowdSorcerer, for example the error rates, how many assignments have been completed and how many peer reviews they have received. The screenshot of the admin UI can be seen in Figure 11; here, assignment does not refer to an individual assignment a student has created, but to a specific assignment instruction (an instance of CrowdSorcerer). Each student generated assignment (called exercise in the backend and the admin UI) has an ID that can be found by inspecting an individual assignment (clicking *Show* button in the right hand side of the row). For anonymity, usernames have been changed – in the actual UI, the names are links that lead to the user's profile.



Assignment

Make an animal guessing game. Ask the user to guess an animal when the correct answer is a cat. If the user guesses correctly, print "Meow!". Otherwise, print "Wrong! Try again."

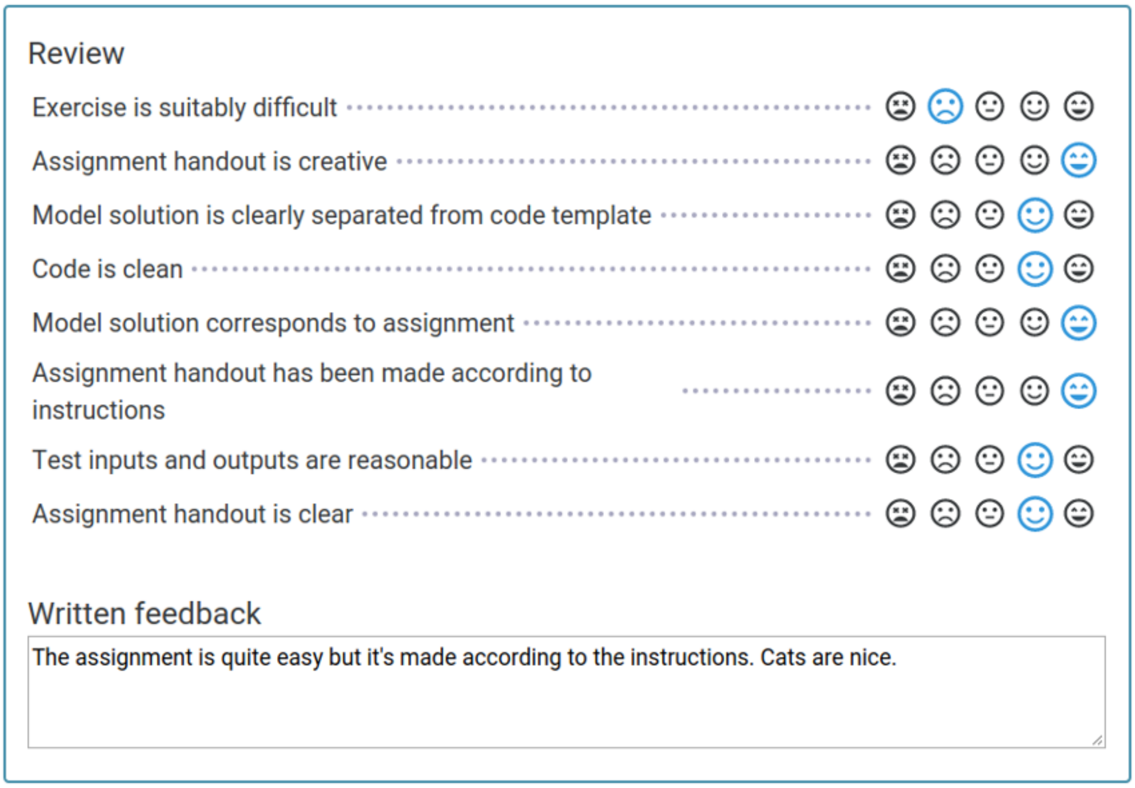
Source code

```
import java.util.*;
public class Submission {

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

        System.out.print("Guess what animal I am thinking:");
        String guess = sc.nextLine();

        if (guess.contains("cat")) {
            System.out.print("Meow!");
        } else {
            System.out.print("Wrong! Try again.");
        }

    }
}

```

Tests

cat	→	Meow!
cow	→	Wrong! Try again.
collared peccary	→	Wrong! Try again.

Figure 9: The basic peer reviewing view of CrowdSorcerer. The reviewer can switch between the code template and the model solution to make comparing these easier. The tool also allows the reviewer to download both the model solution and code template separately to run on their own computer if they want to try out the assignment themselves.



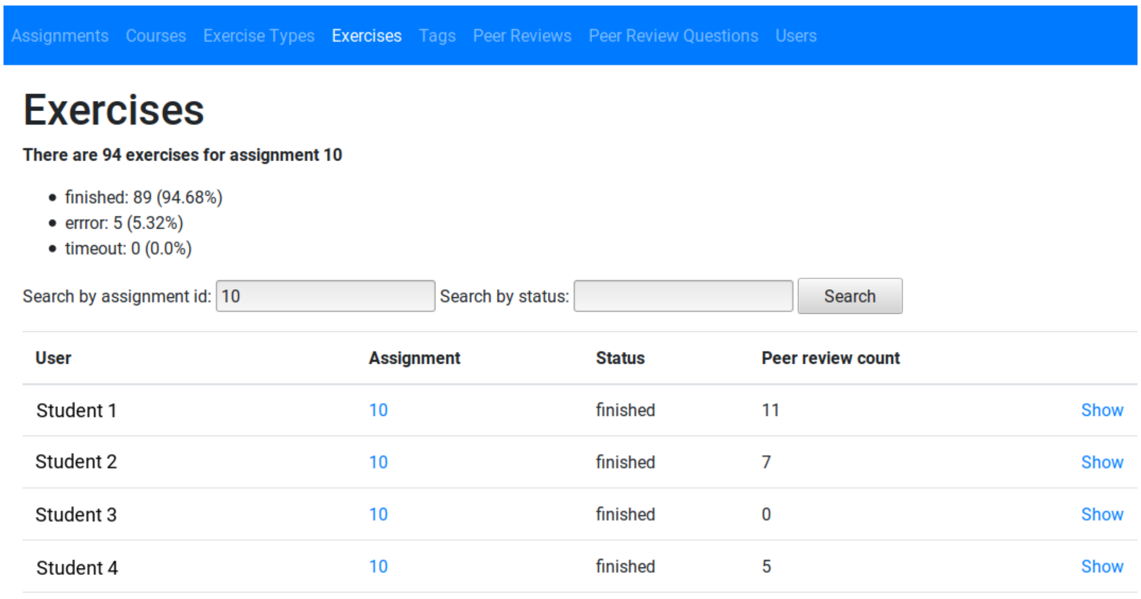
Review

Exercise is suitably difficult	😫 [✓ ☹] 😐 🙂 😄
Assignment handout is creative	😫 ☹ 😐 🙂 [✓ 😄]
Model solution is clearly separated from code template	😫 ☹ 😐 [✓ 🙂] 😄
Code is clean	😫 ☹ 😐 [✓ 🙂] 😄
Model solution corresponds to assignment	😫 ☹ 😐 🙂 [✓ 😄]
Assignment handout has been made according to instructions	😫 ☹ 😐 🙂 [✓ 😄]
Test inputs and outputs are reasonable	😫 ☹ 😐 [✓ 🙂] 😄
Assignment handout is clear	😫 ☹ 😐 [✓ 🙂] 😄

Written feedback

The assignment is quite easy but it's made according to the instructions. Cats are nice.

Figure 10: Review statements and the given Likert scale. Students are required to give a short, written feedback in addition to the grading.


Assignments Courses Exercise Types Exercises Tags Peer Reviews Peer Review Questions Users

Exercises

There are 94 exercises for assignment 10

- finished: 89 (94.68%)
- error: 5 (5.32%)
- timeout: 0 (0.0%)

Search by assignment id: 10 Search by status: Search

User	Assignment	Status	Peer review count	
Student 1	10	finished	11	Show
Student 2	10	finished	7	Show
Student 3	10	finished	0	Show
Student 4	10	finished	5	Show

Figure 11: Exercise page in the admin UI.

3.2 Data collection

The data studied in this thesis was collected in the autumn of 2018 during the second week of an introductory Java programming course in the University of Helsinki. The course consists of a total of seven weeks, and teaches the typical introductory Java programming topics, such as variables, conditionals, loops, functions, objects, and object-oriented programming. The course uses an online textbook which contains integrated programming exercises and other practices, CrowdSorcerer included. The programming exercises are generally small, so the students complete some tens of exercises each week, as opposed to completing fewer, larger projects. The recommended programming environment for the course is NetBeans[16]. This iteration of the course also had weekly lectures, as well as walk-in laboratories for added support.

The data was collected from 91 students who gave their permission for using their data for scientific purposes, and completed an assignment using CrowdSorcerer. The instruction for the CrowdSorcerer assignment for the second week was the following:

"Create an assignment that requires a student to create a program that reads an integer from the user, uses a conditional statement to inspect the integer and then prints a string. For tests, give an example input and the output the program will print with this input."

During the first weeks of the course, there was a bug in the CrowdSorcerer's backend that resulted in erroneously creating empty test files within certain conditions. This allowed some of the assignments to reach the finished state (all tests passed) even though there were some errors in the code, and in regular conditions, the student would have been expected to fix these errors in order to complete the task.

Out of the 91 assignments, 13 had errors in them – however, due to the aforementioned bug, the system only marked three of these as errored assignments, and passed 10 of them, even though an error message was shown to the students. Normally, errored assignments are not shown in the peer reviewing phase, but these 10 were marked as finished by the system, so they received a few peer reviews each. These assignments were not excluded from the study, and were investigated more closely to see whether the peer reviewers were able to notice the errors in the assignments without being prompted to specifically look for any code-breaking errors.

There was only one properly finished assignment that did not receive any peer reviews at all, even though it was completed correctly. Generally, this should be a rare case, as the backend of the system has a drawing logic for the distribution of peer reviews to ensure somewhat equal distribution of peer reviews to assignments. A completed assignment without any peer reviews occurs most likely due to very late return of the assignment, or severe lack of motivated reviewers. In addition, the three assignments that errored properly and did not receive a finished status due to the bug did not receive any peer reviews either. The finished assignments from the second week of the course received 409 peer reviews in total, ranging from 2 to 12 peer reviews per assignment.

[16]https://netbeans.org/

3.3 Research design

The research questions for this thesis are as follows:

- **RQ1.** What types of assignments do students create?
- **RQ2.** How do students perceive crowdsourced programming assignments?
- **RQ3.** To what extent experts and students are in agreement with their reviews?
- **RQ4.** What characteristics are there in assignments in which experts and students agree the most?
- **RQ5.** What characteristics are there in assignments in which experts and students disagree the most?

To answer RQ1, the set of 91 assignments was categorized according to their types and features. The categorisation identified seven different categories based on the simplicity-difficulty level of the assignments.

To answer RQ2, the overall peer review grades were inspected using simple statistical methods. This was to reveal general conceptions students have about crowdsourced assignments, and how students tend to grade their peers overall. Though the written feedback was not reviewed systematically, a tentative overlook to the feedback collected is also included.

To answer RQ3, the assignment set was expert reviewed manually by the author using the same review statements as in the peer reviews. The peer reviews and expert reviews were then compared using inter-rater reliability, more specifically Krippendorff's alpha. This analysis is divided into two sections: reliability by statement and reliability by assignment. For the by statement analysis, each grade given in the expert review for each statement is compared to the grades given to the same statement in peer review, as is shown in Figure 13. For the by assignment analysis, all the grades across all the review statements given in the expert review are compared to those given for the same assignment in peer review, as seen in Figure 14. This review setup is also illustrated in Figure 12.

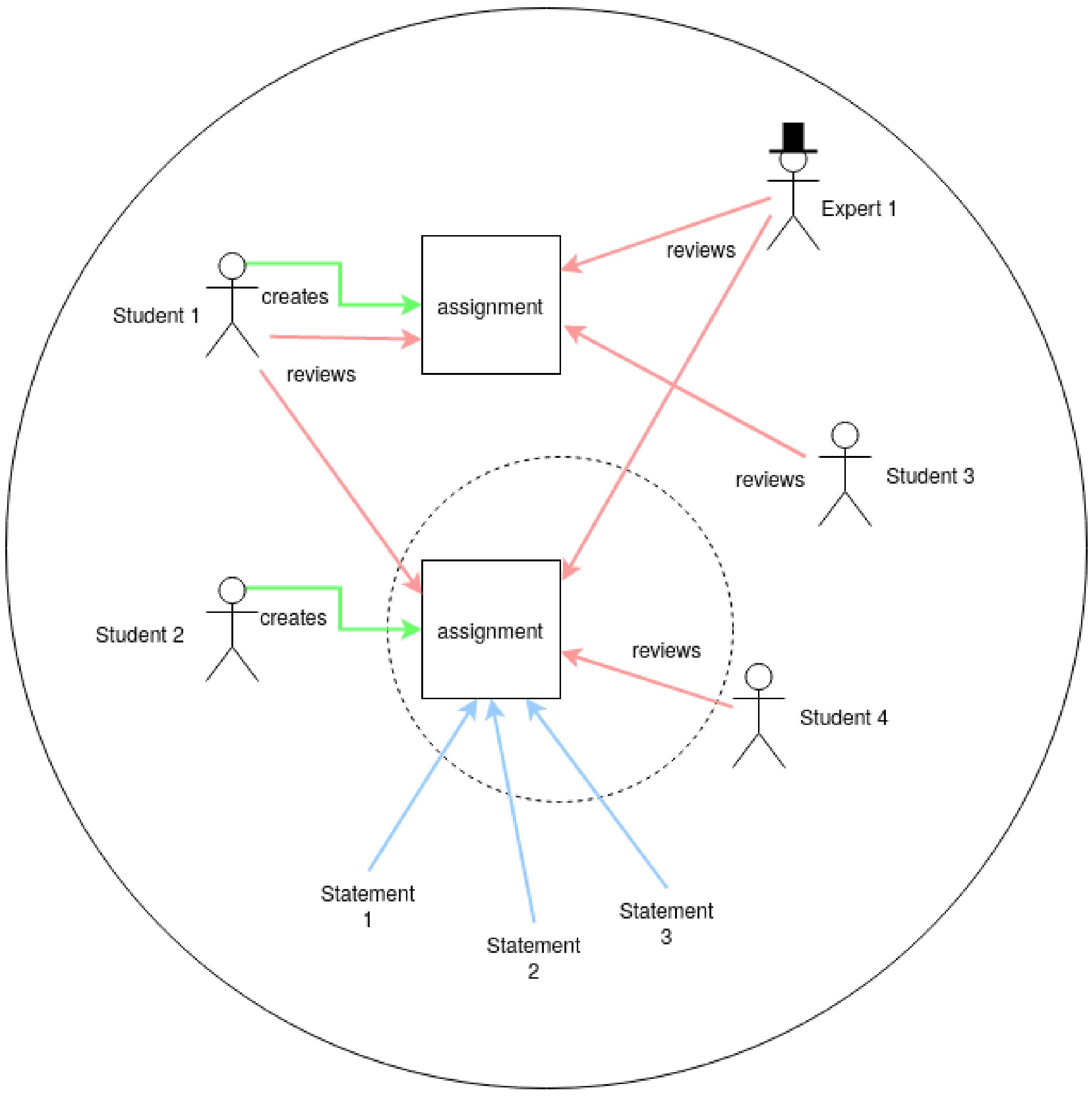
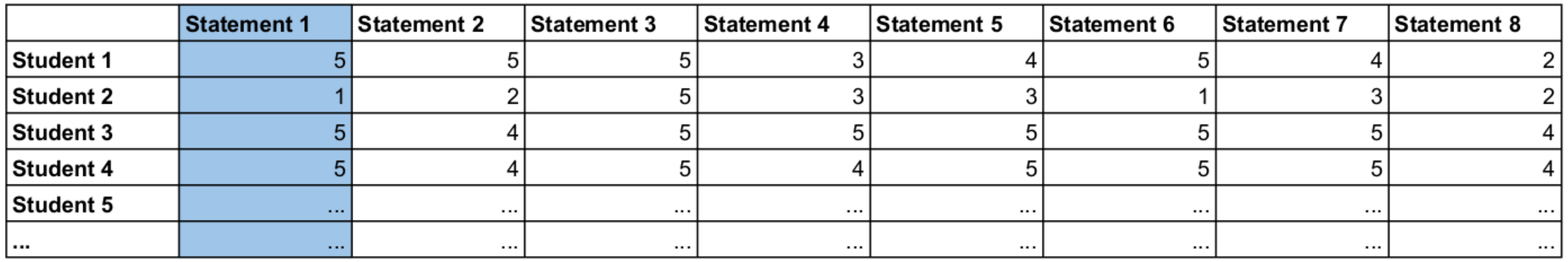


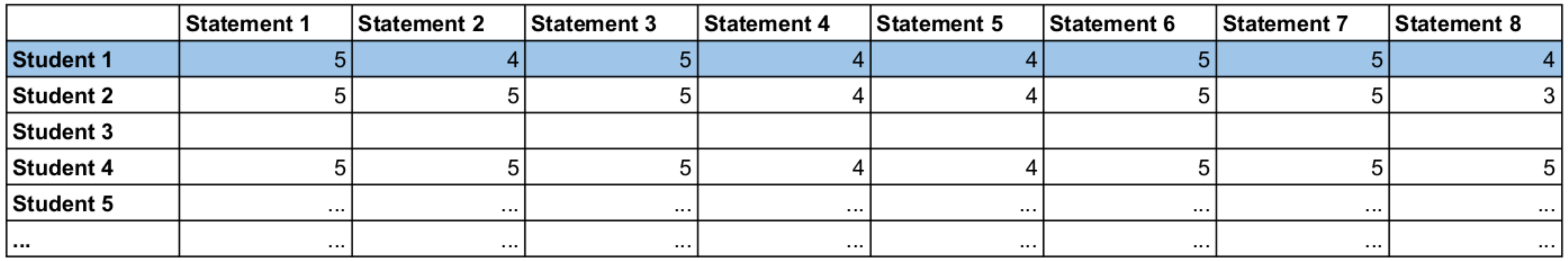
Figure 12: Students can both create one assignment for each instance of CrowdSorcerer, and review multiple assignments, though some choose to only create or only review. An expert reviews all the assignments using the same statements as the students in their peer reviews. The agreement calculations between the peer and the expert reviews are conducted by statement and by assignment.

To answer RQ4 and RQ5, assignments that received the highest and the lowest alpha values (as calculated in RQ3) were further studied to find defining characteristics that may have impacted the reviews, both for the peer and the expert reviews. For RQ4, the purpose is to define characteristics that may commonly occur in a good student-created programming assignment, while for RQ5, the purpose is to find possible reasons why the expert and the peer reviews have such noticeable disagreements.



	Statement 1	Statement 2	Statement 3	Statement 4	Statement 5	Statement 6	Statement 7	Statement 8
Student 1	5	5	5	3	4	5	4	2
Student 2	1	2	5	3	3	1	3	2
Student 3	5	4	5	5	5	5	5	4
Student 4	5	4	5	4	5	5	5	4
Student 5	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 13: Example of the expert review data table. For the purpose of anonymity, students' usernames have been replaced with Student 1, Student 2, ... The column with the blue background demonstrates the first inter-rater reliability calculation (by statement) - these grades were compared to the same column of the peer review data table.



	Statement 1	Statement 2	Statement 3	Statement 4	Statement 5	Statement 6	Statement 7	Statement 8
Student 1	5	4	5	4	4	5	5	4
Student 2	5	5	5	4	4	5	5	3
Student 3								
Student 4	5	5	5	4	4	5	5	5
Student 5	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 14: Example of the peer review data table. For the purpose of anonymity, students' usernames have been replaced with Student 1, Student 2, ... The row with the blue background demonstrates the second inter-rater reliability calculation (by assignment) - these grades were compared to the same row of the expert review data table. Since Student 3 did not receive any peer reviews, their row is empty. Their assignment has been removed from the comparison analysis, even though it was expert reviewed.

3.4 Expert review grading criteria

The statements used for both the peer and expert reviews were originally created for the initial version of CrowdSorcerer. The statements were chosen so that they inspect the quality aspects of the assignment from various perspectives as comprehensively as possible. As of the time of writing this, some review statements have been modified – see Section 5.3 for discussion.

In order to remain consistent in the expert reviews, a set of grading criteria was established and followed throughout the review process. Though the criteria are based on the software quality standards presented in Section 2.2, strict specifications are not followed. This is because industry standards would be too harsh for novice programmers who haven't been taught the quality aspects yet. The reviewing process through the students' point of view is described in Section 3.1.

The model solution corresponds to the assignment handout: The model solution created by the student is expected to match the instructions they give in the assignment handout. Generally, this statement was well-graded, as vast majority of the assignment handouts matched perfectly to the suggested model solution, or had very minor issues, resulting in a subtraction of one point. The bigger the discrepancy between the assignment handout and the model solution, the more points were taken from the full five point marks.

The code is clean: Cleanliness of a program code describes its readability, adherence to coding practices taught on the course and general best practices for the programming language used. Measuring the cleanliness of a program is very much possible with systematic methods [68], and these methods were roughly applied in the expert review process.

The model solutions were graded so that each started with the base of five points, which is the maximum. Points were subtracted according to the amount and severity of the style issues – inconsistent use of spaces or line changes, unconventional placement of brackets, and confusing order of declared variables being under scrutiny, to mention a few. There were generally very few issues with the readability and overall cleanliness of the code, as the exercises were short and usually very similar to those on the course.

The model solution and the code template are separated correctly: While originally this statement was meant to measure how well the students can identify the aspects that are needed for the code template and what is necessary for the model solution part of the program, it ended up serving a dual purpose of measuring how well the students can use CrowdSorcerer. In the original version of the tool, there were some issues with the marking of the model solution and code template lines, namely that students tended to forget the whole process so that the model solution and code template ended up identical. This issue was alleviated in the version that was used in the course iteration this data is collected from by using a preview window that shows the model solution and code template before sending, and urges the student check that these have been marked correctly.

It seems that the preview window did help with the separation issue, as there were no assignments with identical model solution and code template. These assignments would have received one point from the expert review out of the five possible points. Assignments that received two or three points were usually cases where the checks would not have passed if it hadn't been for the bug in the system, or the assignments were actually left in the error state and not finished. Four points were given to those assignments that were correctly marked, but left too little for the user to do to make the exercise reasonable – that is, had too many lines in the code template. Five points were given to assignments that had reasonably marked model solution lines and good balance between the solution and the template.

The assignment is creative: Creativity of a programming assignment is a very subjective matter, and the grading is likely to depend on the sense of humour, personal interests and background knowledge of programming exercises for each reviewer. The points were awarded as follows:

- One point given if the assignment is directly plagiarized from the example given in the instructions on how to use CrowdSorcerer. As the assignment handout, model solution, code template and tests are fully given in the instructional video, this option does not require any individual work from the student besides writing out the given solution to their instance of the tool.
- Two points for assignments that directly copy an exercise from the course material. Though not optimal for a learning experience, this still requires that the student has completed the exercise, either while using the tool or beforehand, as they need to provide the model solution.
- Three points for assignments that are heavily inspired by an exercise in the course material, though not directly copied. These can be, for example, cases where the variables are slightly changed, even though the structure is identical to the exercise on the course. These kinds of assignments are expected, as most of the solutions are very simple and rarely novel.
- Four points for assignments that use very simple solution for an unique topic to create an interesting and novel assignment.
- Five points for a novel topic or implementation idea. Usually more complex in their model solution than the assignments that were awarded four points.

It should be noted that while copying is very much against good practices, it was not explicitly prohibited for the CrowdSorcerer assignments. The students were encouraged to take inspiration from the course exercises if they had trouble coming up with their own ideas. Students who directly copied their assignment from the CrowdSorcerer example video or the course materials were not sanctioned in any way.

The assignment is suitably difficult: When examined subjectively, the difficulty of the assignment depends heavily on the knowledge and experience level of the student or user. Objectively, the difficulty level of the assignment should depend mostly on the topics that have been covered so far during the course. In more complicated exercises, added difficulty may arise due to the structure and requirements of the exercise, but in simpler exercises, the difficulty is provided by the topics used on the exercise. For example, using recursion is perceived more difficult than using simple if-else statements.

The instructions for the assignment asked to use a conditional statement, take in integer input from the user using Scanner and print out string output. The very basic implementation of this awarded the student three points in the expert review, as the course had advanced beyond these topics and the students are capable of using other tools beside these. Four to five points were awarded for example for the correct use of relational operators, for-loops and while-loops. Grades of one to two points usually had other issues that also affected their difficulty indirectly, such as omitting the use of conditional statement completely.

The assignment handout corresponds to the instructions: This statement requires that the assignment handout written by the student corresponds to the instructions given by the lecturer. For example, if the instructions require using if-else statement, the assignment handout given directly or indirectly requires the user completing the assignment to use if-else statement. Though direct instructions make grading this clearer, indirect instructions are not an issue if the assignment handout is clearly written (reviewed in the next statement). Indirect instructions in the handout may actually provide a slight increase in difficulty of the assignment.

The grading started with five points and points were subtracted on a case-by-case basis depending on how far from the lecturer's instruction the student drifted. Completely disregarding the instructions awarded three points or less, while smaller mistakes such as asking for a string input instead of integer subtracted one point.

The assignment handout is clear: The clarity of the assignment handout is a sum of multitude of things. A clear handout should, first and foremost, have all the information needed to actually complete the exercise – variables, parameters and clear outputs. In more complicated exercises, the handout should give a hint of the actual structure of the expected program, though some room needs to be left for the user's own thought process.

Besides the obvious necessity of needing all the pieces required to complete the exercise, a clear assignment handout is also linguistically coherent. The words used should match the vocabulary of the course, and the sentences used should not be overtly long to easily portray the purpose and the goal of the exercise. While grammatical issues, lacking punctuation and typing errors rarely affected the grading, the cases where the assignment handout becomes very difficult to read required some subtractions to the overall score for this statement.

The expert review gradings for this statement started from the full five points, and subtracted points cumulatively approximately as follows:

- Subtract one point if the assignment handout is missing information.
- Subtract one point if the assignment is very difficult to understand because of grammatical issues.
- Subtract one point if the assignment handout gives wrong or misleading instructions.

Assignments that received only one point from this statement's grading were generally very vague, and it would be impossible to complete the exercise with the information given.

The test cases are reasonable: Though there are very concrete and studied ways to determine test coverage and quality [68], these methods are excessive for the small assignments created with CrowdSorcerer. They would also most likely be too harsh, considering that the students are on their second week of the introductory programming course and have not been introduced to the topic of testing yet.

The instructions for the tool noted that the students should consider all the paths their program can take, and test at least all the possible outputs their solution can produce. Thus, the grading was heavily based on the test coverage, namely how precisely the test cases represent the branches of the conditional statements. The grading was a bit more fluid, as the required test cases vary depending on the implementation of the assignment.

The assignments that received one or two points had major flaws in their test cases. Most commonly, several outputs were not tested, and in some cases, the assignment was actually flawed so that it would not have passed the checks if proper tests were implemented. The assignments with three points had some flaws, but covered most of the test cases deemed necessary. The test cases used covered all the flaws the model solution could have so that the solution itself is checked, though some user inputs may still cause unwanted behaviour. The assignments that received four points had some minor flaws, usually concerning boundary values when the model solution uses relational operators. The assignments with five points had perfect test coverage on the level expected, and usually demonstrated ability to create test inputs and outputs in a systematic way, for example, starting from negative values and moving towards positive values, acknowledging possible boundary values in between.

4 Results

The following subsections present the results for each of the five research questions in the same order as described in Section 3.3.

Category	**Students**
Only printing	2
Single if	2
Multiple ifs	2
Simple if-else	54
Intermediate if-else	16
Advanced if-else	8
Loops (+ if-else)	7

Table 1: Categorized assignments, 91 in total.

4.1 Categorizing student-created assignments

To answer the first research question, *"What types of assignments do students create?"*, the student-created programming assignments were categorized according to their features. Since the instructions prompted for an assignment that asks the user for an integer input, uses a conditional statement and prints a string output, the simplest expected assignments should have at least one if-statement.

The categorized assignments can be found in Table 1. In this table, the categories are as follows:

- **Only printing:** 2 assignments. The assignment does not include any conditionals, loops, or other relevant structures – the program only prints hard-coded statements. The Scanner, already declared and locked in the source code as a hint for the students, is not used.
- **Single if:** 2 assignments. The assignment uses a singular if-statement, but no else or if-else. The simplest acceptable assignment type for the instructions given.
- **Multiple ifs:** 2 assignments. The assignment uses multiple if-statements one after another, but not nested or if-elses.
- **Simple if-else:** 54 assignments. Usually a simple if-else with one relational expression. See Example assignment 1.
- **Intermediate if-else:** 16 assignments. Slightly more complicated assignment, such as using more complicated combinations of relational expressions, or multiple if-elses. See Example assignment 2.
- **Advanced if-else:** 8 assignments. The assignment uses complicated combinations of relational expressions, and nested conditionals. See Example assignment 3.
- **Loops (+ if-else):** 7 assignments. The assignment uses some kind of loop in addition to conditional expressions. See Example assignment 4.

The assignments in the first category are automatically faulty from the review point of view, as they do not follow the instructions given. All of the assignments that included loops were either intermediate- or advanced-level regarding the use of conditionals and relational expressions.

The following assignments are examples of the typical program in some of the categories mentioned above. All of the assignments have been translated from Finnish to English by the author. For readability purposes, the assignments have been cleaned up where necessary, and the Java package imports and class declarations have been omitted.

```
//Example assignment 1

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Give an answer: ");

    int answer = Integer.valueOf(sc.nextLine());

    if (answer == 42) {
        System.out.println("But what was the question?");
    } else {
        System.out.println("OK");
    }
}

//Author's note: example of the category simple if-else
```

```
//Example assignment 2

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("How many cats do you have? ");

    int cats = Integer.valueOf(sc.nextLine());

    if (cats == 0) {
        System.out.println("What a pity!");
    } else if (cats >= 1 && cats <= 4) {
        System.out.println("Exemplary work!");
    } else if (cats > 4) {
        System.out.println("Wow, what a herd!");
    }
}

//Author's note: example of the category intermediate if-else
```

```
//Example assignment 3

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

    if (number % 6 == 0) {
        if (number % 4 == 0) {
            System.out.println("The square of the given number is: "
            + number * number);
        } else {
            System.out.println("The cube of the given number is: "
            + number * number * number);
        }
    }
}

//Author's note: example of the category advanced if-else
```

```
//Example assignment 4

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int positives = 0;
    int sum = 0;
    int i = 0;

    System.out.println("Please give three numbers: ");

    while (i < 3) {
        int number = Integer.valueOf(sc.nextLine());

        if (number > 0) {
            positives++;
            sum += number;
        }

        if (positives == 0) {
            System.out.println("The average cannot be calculated");
        }

        System.out.println("The average is: " + 1.0 * sum / positives);
    }
}

//Author’s note: example of the category loops (+ if-else)
```

4.2 Student perceptions of assignments

To answer the second research question, *“How do students perceive crowdsourced programming assignments?”*, analysis was mostly focused on the grading the students gave each other in the peer review statements. No systematic analysis was done on the written feedback the students were required to give.

On average, students graded the assignments higher than the expert. Over all the statements, the peer reviews average to five points, whereas expert reviews round to four points. Peer reviews also had a much lower standard deviation across all the statements – the average standard deviation for the student-given grades was around 0.41, while the standard deviation for expert reviews was approximately 0.92.

From these results, it seems that students were generally very hesitant to give lower points to their peers – many seemed to subtract points from the full five points only for glaring mistakes. Whether this is because students did not want to grade their peers too harshly or they simply did not notice smaller mistakes in the assignments is unknown, but discussed further in Section 5.

While the open feedback was not inspected systematically for this study, a cursory glance to it did reveal some general trends. Most of the time, students seem to be very succinct in their feedback. The minimum for the open feedback was three words, and while some of the students did give proper feedback, there was a noticeable group that gave the most minimalistic answers possible, sometimes circumventing the word count minimum with answers such as "Very nice ." (period separated from the last word to increase the word count). However, since the open feedback was not systematically reviewed, no conclusions are drawn from these observations. Some ideas for future work on the topic are presented in Section 5.3.

4.3 Agreement

The following subsections present the results of inter-rater reliability calculations for each statement (Section 4.3.1) and each assignment (Section 4.3.2). The used measurement is ordinal Krippendorff's alpha as per description in Section 2. These sections answer the third research question, *"To what extent experts and students are in agreement with their reviews?"*.

It should be noted that since the students were able to create a maximum of one assignment each, the words *assignment* and *student* can and are sometimes used interchangeably in these subsections. Also, the inter-rater reliability calculations use only 87 assignments out of the set of 91 assignments, as four of the assignments did not have any peer reviews, and thus, could not be compared to the expert review.

4.3.1 Reliability by statement

When Krippendorff's alpha is measured for each statement so that the peers are considered as one rater and their ratings are compared to the expert's ratings, the inter-rater reliability values are very low. The results of this calculation can be seen in Table 2.

To categorize the results in Table 2 according to the magnitude guidelines by Landis and Koch [95]:

- Fair agreement (0.21-0.40): 1 statement
- Slight agreement (0-0.20): 5 statements
- Agreement random or worse (less than 0): 2 statements

The average agreement is $\alpha = 0.09$.

As seen above, the inter-rater reliability only reaches fair agreement at its best. This was for the statement "The assignment is suitably difficult". Most of the statements fall in to the category of slight agreement, which is the lowest possible category without being completely random. Two of the statements, "The model solution and

Review statement	α
The model solution corresponds to the assignment handout	0.17
The code is clean	0.10
The model solution and the code template are separated correctly	-0.10
The assignment is creative	0.08
The assignment is suitably difficult	0.30
The assignment handout corresponds to the instructions	0.02
The assignment handout is clear	0.15
The test cases are reasonable	-0.03

Table 2: Ordinal Krippendorff's alpha for each statement over all the assignments/students.

the code template are separated correctly" and "The test cases are reasonable", got a negative alpha value, meaning that the agreement for these statements was random or worse. The implications of these results are discussed in Section 5.1.3.

Even with the most lenient minimum acceptable alpha values [95], these results can be used to draw tentative conclusions at best. Calculating the inter-rater reliability over statements is not a reliable way of using the alpha coefficient, as using all the peers as one singular rater introduces too much noise to the data.

4.3.2 Reliability by assignment

Table 3 in Appendix 1 shows the ordinal Krippendorff's alpha values for each student's assignment. All the peer reviewers for each assignment are treated as one rater, and the grades given by them are compared to those given in the expert review. To categorize the results in Table 3 according to the magnitude guidelines by Landis and Koch [95]:

- Almost perfect agreement (0.81-1): 3 assignments
- Substantial agreement (0.61-0.80): 7 assignments
- Moderate agreement (0.41-0.60): 19 assignments
- Fair agreement (0.21-0.40): 24 assignments
- Slight agreement (0-0.20): 18 assignments

- Agreement random or worse (less than 0): 16 assignments

The average agreement is $\alpha = 0.27$.

Though the majority of the assignments do get categorized to slight to moderate agreement, the alpha coefficient gives more reliable results when used over students and their assignments instead of statements. In this case, the peer reviews come from a handful of students, the exact number depending on how many peer reviews the assignment received, ranging from 2 to 12 peer reviews. Thus, the dispersion between peer reviews does not cause as much effect as it does with calculations in Section 4.3.1. However, a noticeable portion of the assignments still got categorized as random or worse agreement, meaning that the results should only be used to draw tentative conclusions.

4.4 Assignment characteristics

Finally, this subsection studies the characteristics of the student-created programming assignments, answering the fourth and the fifth research questions, *"What characteristics are there in assignments in which experts and students agree the most?"* and *"What characteristics are there in assignments in which experts and students disagree the most?"*. It should be noted that these assignments are not the best and the worst rated out of the data set, and should not be considered examples of such. The assignments used as examples here were simply rated in a way that the agreement was notably high or low between the peer and the expert reviews.

As before, all the assignments used as examples have been translated from Finnish to English by the author, and may have been slightly modified for readability purposes. The alpha values used are specifically from the reliability by assignment calculations of Section 4.3.2.

Three assignments were graded so that the Krippendorff's alpha value is above 0.80 agreement, meaning almost perfect agreement between the raters. These assignments were by Student 66 ($\alpha = 0.96$), Student 49 ($\alpha = 0.95$) and Student 8 ($\alpha = 0.82$) (Table 3 in Appendix 1). All these three assignments were neatly written, both in code and handout, and had no issues with their model solutions. Coincidentally, these assignments were also from the simpler end of the assignments created during the first week, so both the expert and the peer reviews took some points out of creativity and proper difficulty. Example assignment 5 shows one assignment that received a high alpha value from the agreement calculation.

To compare, the lowest agreement values were given to assignments by Student 35 ($\alpha = -0.43$), Student 55 ($\alpha = -0.50$) and Student 77 ($\alpha = -0.63$) (Table 3 in Appendix 1). Since negative values are considered too divergent from the reasonable reliability values to matter [91], all of these assignments could be treated as equally in disagreement. The lowest values that were not in the negatives were rounded to zero, and thus would not have provided effectively different comparisons. Two out of the three assignments were labeled as finished only due to the bug described earlier, and they should not have passed the tests. Thus, these two assignments received lower points from the expert review, while the peer reviews gave the assignments relatively good grades. Example assignment 6 shows one assignment that received a low alpha value from the agreement calculation.

In total, there were ten assignments that passed only due to the aforementioned bug. Categorizing them separately with the guidelines by Landis and Koch [95]:

- Moderate agreement (0.41-0.60): 2 assignments
- Fair agreement (0.21-0.40): 4 assignments
- Agreement random or worse (less than 0): 4 assignments

The errored assignments received generally poor agreement rates, majority of the assignments falling into fair or random agreement categories. Generally, even if students were able to recognize something amiss in an assignment, their judgement was not as strict as in the expert review, and most of the time, they were not able to recognize all the mistakes in the assignment. This is expected, as the students were not explicitly told that the reviewable assignments may contain fatal errors, so the students were not specifically looking for these types of characteristics while reviewing.

```
//Example assignment 5

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Which month were you born in?
                        Enter a number! ");

    int month = Integer.valueOf(sc.nextLine());

    if (month == 12 || month == 1 || month == 2) {
        System.out.println("Winter child");
    } else if (month == 3 || month == 4 || month == 5) {
        System.out.println("Spring child");
    } else if (month == 6 || month == 7 || month == 8) {
        System.out.println("Summer child");
    } else if (month == 9 || month == 10 || month == 11) {
        System.out.println("Autumn child");
    } else {
        System.out.println("Not a month!");
    }
}

//Author's note: Example assignment from the highest alpha values
```

```
//Example assignment 6

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter a number: ");

    int number = Integer.valueOf(sc.nextLine);

    if (number < 0) {
        System.out.println("Freezing!");
    } else {
        System.out.println("Whew, too hot");
    }
}

//Author's note: Example assignment from the lowest alpha values.
  This was an errored assignment, as the required brackets are
  missing after Scanner nextLine() method declaration.
```

5 Discussion

In the following subsections, Section 5.1 discusses the implications of the results of each research question, proceeding in the same order as Section 4. Section 5.2 reflects the limitations this study has, and lastly, Section 5.3 considers future directions, taking into account both the further development of CrowdSorcerer and the possible future studies.

5.1 Analysis of the results

5.1.1 Categorizing student-created assignments

To answer the first research question, *"What types of assignments do students create?"*, the student-created programming assignments were categorized according to their most notable features. The vast majority of the assignments fall into the first four categories, meaning that most of the assignments collected are very simple both in structure and in difficulty (see Table 1 in Section 4.1 for the categories). In total, 60 of the 91 assignments are categorized as *simple if-else* or simpler. This is expected, as the majority of the students on the course are novices as they start, and assuming more complicated programs from the majority of them would be unreasonable. Also, there are only so many ways one can implement the assignment according to the instructions with the tools the students have learned during the first two weeks of the course. Since completing CrowdSorcerer assignments did not award any points during this iteration of the course, it is also very likely that many students did not want to spend too much time on the tool, and would rather use their effort on exercises that do affect their grade. Thus, even students who perhaps could have been able to create more complicated programming assignments might have chosen to implement as simple programs as possible.

While the program structure for many of the simple assignments was exactly the same as some example programs in the course material, most of the assignments created in CrowdSorcerer did change the topic to something novel. There was one type of assignment that appeared frequently, *"tell the user whether the number they gave is even or odd"*. Though this had also appeared in the course material as a practice exercise during the first week, there were no sanctions for copying or re-implementing previously used code, as it was deemed better that the students at least try the tool and practice input-output testing in the process.

This study did not inspect whether those who created intermediate or more complicated assignments (31 out of 91) had more programming experience at the beginning of the course, or if their grades or general completion rates for each week differ from those who decided to create simpler assignments. The possibilities of a study like this are discussed in the Section 5.3. Also, as this study uses data only from the very beginning of an introductory programming course, these results may not be applicable in cases where the students have more advanced programming skills, but are also required to complete more complex tasks with CrowdSorcerer.

5.1.2 Student perceptions of assignments

To answer the second research question, *"How do students perceive crowdsourced programming assignments?"*, the grades the students gave each other in the peer review statements were analysed. Looking at the ratings across all the statements, peer reviews average to 5 points, whereas expert reviews average to 4 points. The difference is most noticeable in the statement "The assignment is suitably difficult". In this case, it is possible that the novice students are not yet very well-versed in analyzing whether a programming assignment is too easy or too difficult objectively based on the course content thus far, and that the subjective answers cause this difference.

Studies about the reliability of peer reviewers in regards to the accuracy of the grading have given mixed results. Some studies note that students tend to undermark their peers [9, 97, 98], while others report slight overmarking [99]. Some studies suggest that as students gain more experience, they give more accurate reviews, or even shift towards undermarking [100].

One of the possible reasons for overmarking might be that students do not want to hurt their peers' grades, even if it is clearly stated that the peer review grades do not affect the course grade. In the context of this study, it is also possible that students give each other better grades than experts simply because they do not notice all the mistakes. There are no model solutions to which the reviewable assignments could be compared, and if the students do not try to complete and run the assignments themselves, it is fairly easy for an untrained eye to miss errors and faulty handouts.

When self-evaluating, students can also tend to give too high grades for the very worst of the exercises, and not good enough grades for the very best [100]. This leads to the grades falling into narrower range than those from the expert reviews. This corresponds with the results of this study – peer reviews have lower standard deviation than expert reviews (0.41 and 0.92, respectively).

5.1.3 Agreement

To answer the third research question, *"Are experts and students in agreement with their reviews?"*, the inter-rater reliability was measured in two ways – by statement and by assignment. First, each grade given in the expert review for each statement is compared to the grades given to the same statement in peer review (see Figure 13 in Section 3.3, and Section 4.3.1), and secondly, comparing the grades for one student's single assignment (see Figure 14 in Section 3.3, and Section 4.3.2).

To begin with, this section goes through the inter-rater reliability measurements across all the assignments one by one using the results from the by statement calculations. This gives quite a low reliability score, as the peer reviewers have widely varying levels of competence, understanding and even interest for the review process.

The lowest values were for the statements "The model solution and the code template are separated correctly" (-0.10) and "The test cases are reasonable" (-0.03). Negative values imply that there is no effective agreement between the raters, or that the agreement is worse than random. These values can be explained in the specific context of this CrowdSorcerer instance.

With the statement "The model solution and the code template are separated correctly", there were cases where the separation of the model solution and the code template was so erroneous that without the aforementioned bug in the backend, the assignments would not have passed the tests. Students rarely noticed these cases, giving the assignment good reviews for the statement, while the expert review scores were very low in these cases. It is also possible that since the data is from the very beginning of the course, the students are not very honed in approximating what is a good amount of code to be shown in the template and what should be left in the model solution only.

The discrepancy with the statement "The test cases are reasonable" clearly shows that the students have very vague understanding of testing at this point of the course, and even though the expert reviews used quite a lax grading criteria for the tests, the scores were still very much not in agreement. This is expected, as the students have not encountered testing on the course material yet, and are only thinking of the input-output cases of their program, so no coverage or borderline case analysis is expected of them.

Most of the statements indicate only slight agreement. After the negative results, the lowest value is for the statement "The assignment handout corresponds to the instructions" (0.02). This is a statement that may be difficult to evaluate precisely, both for peers and experts, especially without testing the assignment in practice. It is possible that the expert reviews are somewhat stricter in this case, as the expert reviewer has most likely had more practice in reading and completing simple programming exercises than the majority of the students, thus being able to identify missing assignment handout information more easily. There might also be confusion between the statements "The assignment handout corresponds to the instructions" and "The assignment handout is clear" for their differences in meaning.

As for the statement "The assignment is creative" (0.08), the low inter-rater reliability is possible due to the subjective nature of the statement. As the definition of "creativity" is left entirely for the reviewer to decide, the determining factors for the score can be the entertainment value, whether the reviewer has encountered a similar assignment before, the topic, or wording of the assignment handout, just to name a few possibilities.

Similarly to the statement "The test cases are reasonable", the results for statement "The code is clean" (0.10) can indicate that the students do not have clear vision of what is required from clean code. These practices have not been taught on the course, and while the examples given in the material are as clean as possible, students tend to make mistakes and incorrectly remember established syntax practices that have no effect on the functioning of the program. Also, since CrowdSorcerer does not highlight small errors like these as an IDE (integrated development environment) would, and the tool does not have automatic indentation shortcut key like for example NetBeans does, there might be cleanliness errors that would not have happened when programming in an IDE.

One of the statements with surprisingly low agreement was "The assignment handout is clear" (0.15). While it is very possible that the peer reviews do not acknowledge the possible linguistic errors and minor inaccuracies in the description, the hypothesis for this statement was that the agreement would be at least moderate. In this case, the fairly low agreement value is most likely due to the way the inter-rater reliability is calculated across all the peer reviews for one statement.

The same applies for the statement "The model solution corresponds to the assignment handout" (0.17). As this seems like something that the students should be able to evaluate fairly well with the skillset they possess, the low agreement is again most likely due to the calculation method.

The only statement with fair agreement is "The assignment is suitably difficult" (0.29). It seems reasonable that the peer reviews and expert reviews are somewhat in agreement, as difficulty can be considered a matter students can have a grasp of, even if they are novices in the field they are studying. The relatively low agreement rate can be explained with the differing views of difficulty - while the expert reviews try to be as objective as possible, taking into account the exercises and materials that have been introduced at the point of the first CrowdSorcerer instance, perceived difficulty can be very subjective and relies heavily on the student's view of the topic at hand.

It should be noted that all of the previously mentioned agreement values are very low, and that the results from the second calculation allow better base for drawing tentative conclusions. The average agreement for the statements is $\alpha = 0.09$. This is very low, falling into the category of slight agreement. As all the peers are considered one rater in the reliability over statements, the results are understandably weak, as a rater with several possibly disagreeing minds will produce confusing results.

For the second inter-rater reliability calculation, expert review grades were compared to peer review grades of each students' singular assignment. The results of this Krippendorff's alpha calculation can be found from Table 3 in Appendix 1. Again, as in Section 4.3.2, to categorize the results according to the magnitude guidelines by Landis and Koch [95]:

- Almost perfect agreement (0.81-1): 3 assignments
- Substantial agreement (0.61-0.80): 7 assignments
- Moderate agreement (0.41-0.60): 19 assignments
- Fair agreement (0.21-0.40): 24 assignments
- Slight agreement (0-0.20): 18 assignments
- Agreement random or worse (less than 0): 16 assignments

The average agreement for the assignments is $\alpha = 0.27$. While still low, falling into the category of fair agreement, this is notably better than with reliability by statement. In this case, even though students are still considered one rater per assignment, there are less people causing noise in the calculation, as the assignments have twelve peer reviewers at the very most.

The general trend is that the assignments that had very low agreement were graded very well by peers, and received low grades in the expert review, usually due to systematic disagreement on multiple statements. The assignments with fair agreement usually had novel topics and had good ideas, but lost points due to lacking execution, such errors peer reviewers did not notice, insufficient test cases or some minor inclarities in the handout.

In the higher agreement categories, from moderate to substantial agreement, the assignments tended to be very well composed, had less inconsistencies in the handouts, and had novel ideas, but lost some points in the expert review due to lacking test cases, thus causing some disagreement between the peer and the expert reviews. As mentioned before, the vast majority of the students do not have any experience or knowledge about software testing at this point of the course, so it is not surprising that this review statement causes systematic discrepancies in the agreement calculations.

As mentioned in Section 2.3, low inter-rater reliability means that either the scale used is defective, the rating instructions are unclear, or the raters need to be retrained [91]. Addressing all of these points may help with the low and middle tier agreements. Clarifying the review statements with, for example, examples or longer descriptions, could help to unify the answers between peer reviewers. It is very possible that during the later weeks, when the students have been introduced to testing, not only does the agreement on the test cases improve, but also the number and the quality of the test cases increases. This is also supported by an extensive literature review by Boud and Falchikov [100]. Their study notes that it seems that students do tend to get better at self-assessment over time, even though results are somewhat inconclusive. Their conclusion is that experience either diminishes the difference between self review and expert review, or steers towards underrating rather than overrating.

It should also be noted that students can be good reviewers, but also in disagreement with each other. Some of the review statements used in this study are inherently subjective ("The assignment is creative"), and thus, reviewers can have very differing opinions.

5.1.4 Assignment characteristics

To answer the fourth and fifth research questions, *"What characteristics are there in assignments in which experts and students agree the most?"* and *"What characteristics are there in assignments in which experts and students disagree the most?"*, the characteristics of the assignments with the highest and the lowest alpha agreement values were inspected in more detail. As discussed already in Section 5.1.3, students are able to recognize very good assignments, and it seems reasonable that with more experience or instructions, they could be able to recognize weak assignments as well. While the review instructions should not be too strict, as this can affect the students' review process, more detailed review statements and examples of features to take note of could help making the reviews more reliable.

Based on the characteristics of the assignments with the highest agreement, even novice students seem to have the ability to recognise well-made assignments. All of the assignments in the almost perfect agreement category (α between 0.81-1) were on the simpler end difficulty-wise, but had sufficient test cases and were not directly copied from the course materials. The peer and the expert reviews were in agreement that these assignments were all in all very well-rounded programming assignments, but were not the most interesting or novel for their topics.

Most of the assignments with negative agreement (random or worse) were errored, that is, the assignments that should not have passed the compilation tests. This is not only for the three assignments with the lowest agreement as presented in Section 4.4, but also for the other assignments in the negative alpha values (16 assignments in total, as seen in Section 4.3.2). Since students were not told to expect assignments with fatal errors, drawing conclusions from these results is difficult, as it can be safely said that the peer reviewers were not sufficiently trained for the review process. An interesting future research would be to inspect whether the peer reviewers are able to find errored assignments in the midst of correct ones, and whether they recognize the compilation-breaking errors correctly. The most typical errors were unpaired brackets and missing characters, such as missing brackets after method declaration, as well as some other syntax issues like forgotten capitalisations of classes such as *String*. These are all issues that would be more noticeable in a proper IDE which the students are used to programming in, and especially the issue of unpaired parantheses often comes due to incorrectly marked model solution lines. This is a design issue in the tool, and making the difference between the code template and the model solution more clear could alleviate this issue altogether.

The assignments with lowest agreements that were not errored were simply unclear, usually due to multitude of reasons. These assignments had serious flaws in the assignment handouts, had severely lacking test cases that could cause issues when completing the assignment, and were not necessarily created using the instructions. The peer reviewers were generally not able to recognize these types of mistakes very clearly. It is very possible that the students would be able to spot these errors if they tried to complete the programming assignment themselves, as is possible through downloading the code template ZIP file, but as this is not required for the review process, it is most likely seen as a waste of time. Since the expert review was able to point out issues with the assignment handout and other features, it is also possible that the peer reviews would be able to put a note on these too when they gain more experience.

5.2 Limitations

This study comes with a range of limitations, which are discussed in this subsection. Threats to both internal and external validity are considered – internal validity referring to how well other explanations to the results can be ruled out, and external validity to whether the results can be generalized into other contexts.

There are limiting factors considering the source of the data for this study. The data has been collected from a single institution, and even more importantly, from one iteration of an introductory programming course. As the expert reviews were only conducted on the second week of the course, it is possible that the context has an effect on the results. For example, even including later weeks from the course, there could be change in the way the students peer review exercises, as they have had more experiences on what programming exercises look like and how they are typically solved. It is also possible that the way the course is structured affects the results. This particular programming course has tens of small programming exercises integrated into the online learning material for each week, as opposed to some courses that use larger assignments that have been split into smaller tasks. It may be that students used to the latter type of exercises would have very different approach to using the tool.

There is also a participation bias when considering both creating the assignments and the peer reviewing process. Since using CrowdSorcerer was optional, it is likely that the students who decided to use the tool form a differing population from the overall course population. For example, it is possible that more active students were more eager to use CrowdSorcerer, and thus, those who decided to skip the tool altogether were excluded from the data.

It is also very possible that the subjective nature of some of the review statements affected the results of this study. To alleviate this in the future studies, the review statements could be separated to those that have clear, correct answers, and those that are open for interpretation, and then inspected separately.

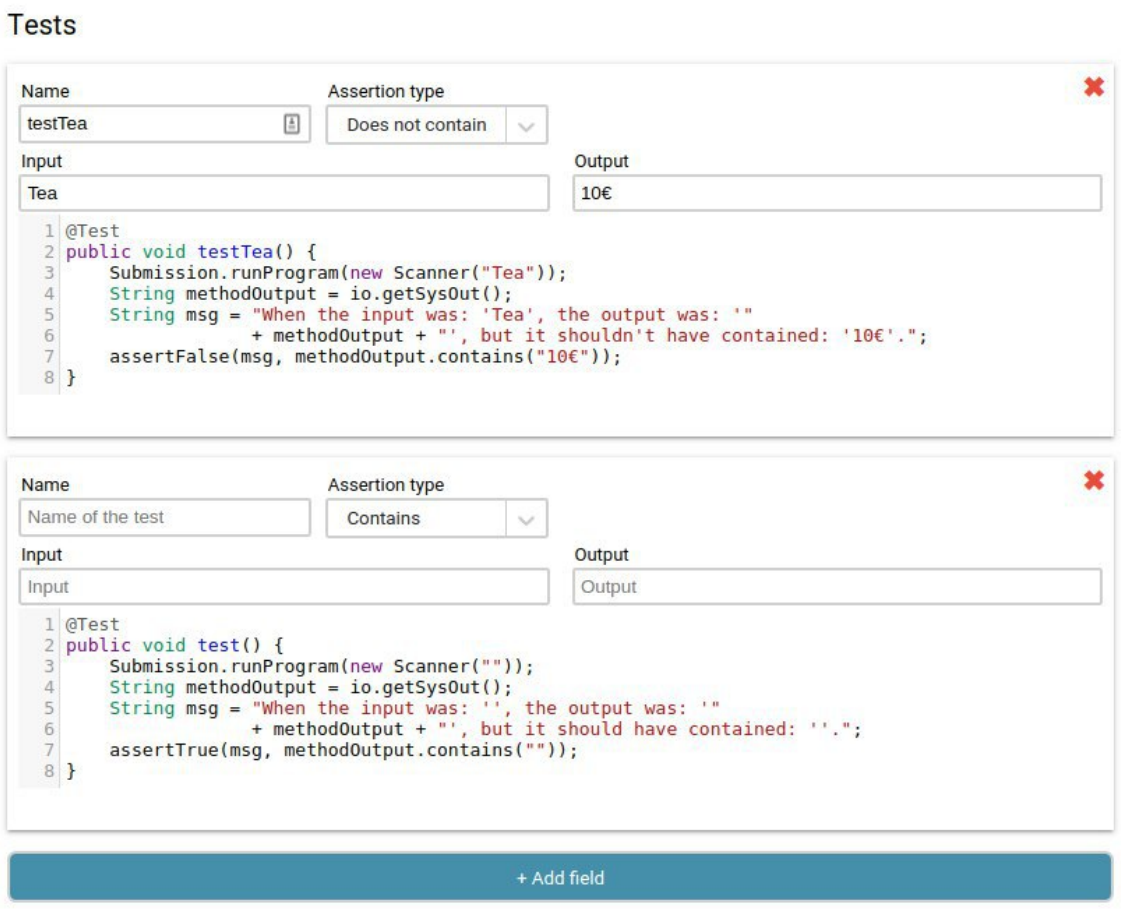
Though the expert review criteria was determined beforehand, it should be acknowledged that the reviewer had only had limited experience in reviewing and grading processes beforehand. Thus, it is possible that the expert reviews are slightly skewed, as the reviewer gained more confidence and insight by the end of the reviewing process. For increased reliability, it would be beneficial to use multiple experts for the reviewing process instead of just one. If all the experts complete all the ratings instead of sharing the reviewing workload, it would also be possible to inspect the inter-rater reliability between experts before analysing the agreement between the experts and the students.

Considering student feedback received after this study was conducted, there may be inconsistencies on how students have understood and interpreted the review statements. Consequently, this can have an effect to the peer review results in general, and in turn, also affect the inter-rater reliability. However, the misconceptions would have to be severe to show up as systematic disagreements and affect the α-agreement, and this was not indicated by the student feedback.

Also stemming from the same student feedback, there were major issues with structuring and usability of CrowdSorcerer. It is possible that these issues have made the assignment creation process needlessly difficult, which in turn can have an effect on the peer reviews. For example, the interface of the tool did not make it clear enough what part of the program was to be the model solution, and what would be the code template. Thus, the review statement "The model solution and the code template are separated correctly" can be very ambiguous for students who struggle to understand the difference between these two altogether. The expert reviews did not take the user interface problems into account during the review process.

5.3 Future work

There are several plans and future directions already decided when it comes to developing CrowdSorcerer, some of which have come to fruition since the beginning of this thesis. For example, the way students are introduced to testing practices has been improved. Now, the students first get the input-output test cases as before, but on the later weeks, they are shown ready-made test methods they need to fill in themselves with provided fields (Figure 15). Finally, when the course material is at the point where testing has been properly taught, the students are expected to write their own test methods as a part of the CrowdSorcerer assignment (Figure 16). The usefulness for learning testing in CrowdSorcerer has been evaluated by Kangas et


Tests

Name: testTea | Assertion type: Does not contain
Input: Tea | Output: 10€

```
@Test
public void testTea() {
    Submission.runProgram(new Scanner("Tea"));
    String methodOutput = io.getSysOut();
    String msg = "When the input was: 'Tea', the output was: '"
                + methodOutput + "', but it shouldn't have contained: '10€'.";
    assertFalse(msg, methodOutput.contains("10€"));
}
```

Name: Name of the test | Assertion type: Contains
Input: Input | Output: Output

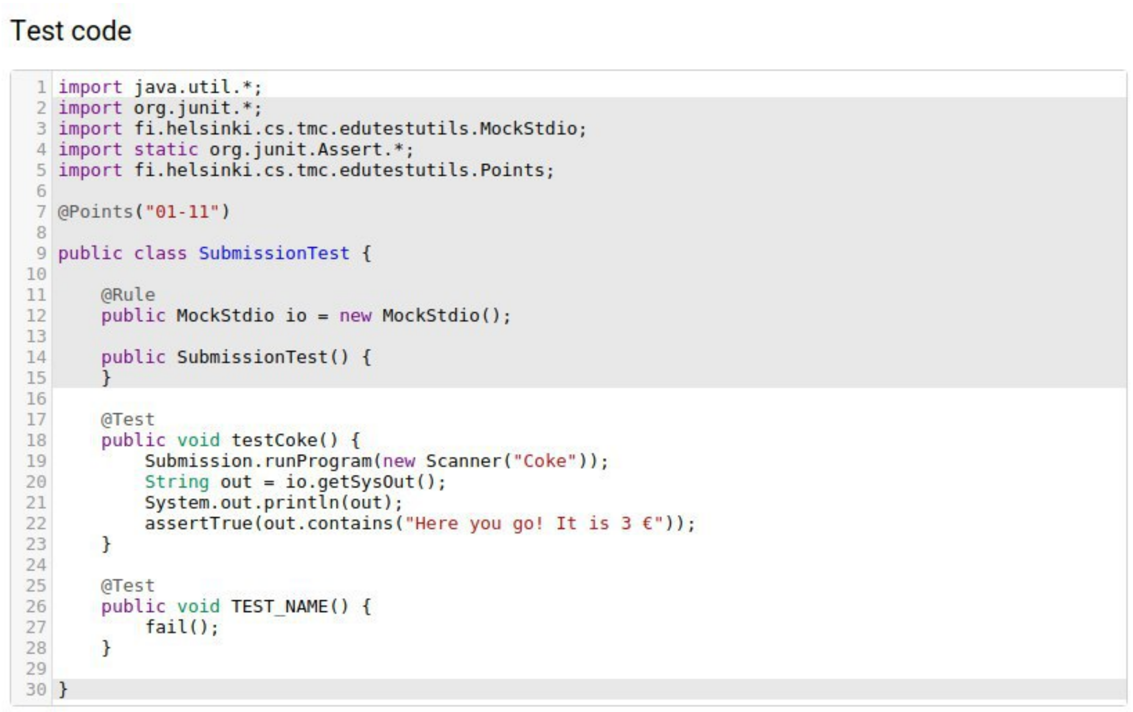
```
@Test
public void test() {
    Submission.runProgram(new Scanner(""));
    String methodOutput = io.getSysOut();
    String msg = "When the input was: '', the output was: '"
                + methodOutput + "', but it should have contained: ''.";
    assertTrue(msg, methodOutput.contains(""));
}
```

Figure 15: New test type in CrowdSorcerer requires the students to fill in the relevant fields of a test method.

al. [13], and further studies with the improved system are in the works.

Besides putting emphasis on using CrowdSorcerer to teach testing, some improvements have been implemented to the tool to alleviate the difficulties in differentiating between the model solution and the code template. When the user submits their assignment, the tool now shows a preview of the model solution and the code template separately, and asks to check that the programs in these two are correct. The wording of the instructions has also been improved. These two changes seem to have made using the tool slightly clearer.

However, through feedback we have received about CrowdSorcerer at the end-of-course questionnaires, the user experience is still somewhat tedious. Most of the feedback concerns are about how slow the submission process is, how the source code field does not work like an IDE or text processor the students are more used to, and how the error messages are unclear. This feedback indicates that the tool might benefit greatly from a user interface redesign, as well as rework regarding how error messages are parsed. The worst of the submission issues have been fixed by


Test code

```java
import java.util.*;
import org.junit.*;
import fi.helsinki.cs.tmc.edutestutils.MockStdio;
import static org.junit.Assert.*;
import fi.helsinki.cs.tmc.edutestutils.Points;

@Points("01-11")

public class SubmissionTest {

    @Rule
    public MockStdio io = new MockStdio();

    public SubmissionTest() {
    }

    @Test
    public void testCoke() {
        Submission.runProgram(new Scanner("Coke"));
        String out = io.getSysOut();
        System.out.println(out);
        assertTrue(out.contains("Here you go! It is 3 €"));
    }

    @Test
    public void TEST_NAME() {
        fail();
    }

}
```

Figure 16: As a final step of using CrowdSorcerer to teach testing, the students are required to write full test methods themselves.

changing the queuing system that handles sending the assignments to the server that runs automatic testing. However, it would be useful if typing mistakes and errors like unpaired brackets could be checked and shown real-time, and only checking for compilation errors would require sending the assignment. This feature is something that is being investigated alongside the interface redesign. As the tool is stable enough, running longer studies, for example, throughout a 14-week period of both the introductory and the advanced course in programming, would provide more data and allow more reliable analyses. Technical issues could also be ruled out with a lab study, in which students are invited to the university's computer laboratory to create CrowdSorcerer assignments with a possibility of asking help from a course assistant.

Since there are plans to launch CrowdSorcerer in the near future in other universities besides the University of Helsinki [14], the tool will require some language updates – both for the natural and programming languages. The interface of the tool has been translated into English, and support for Python assignments has been added. As these features are developed, upgrading the tool for different localisations and programming courses becomes easier. This would also grant access to different contexts, and cross-institutional studies.

Updating the basic set of the review statements should be considered in the future. Although new statements have been added, usually depending on the types of assignments, the basic set has stayed the way it has been since the beginning. Rewording statements based on the student feedback and considering how the basic set could be improved so that it delivers an overview to the quality of the assignments as comprehensively as possible would increase the reliability of the reviews. This could be done, for example, by referencing existing review rubrics [80].

Analyzing the written feedback would grant a clearer view into the peer reviews, and the reviewers' thoughts. Though the brief exploration revealed that some students opt for shortest answers possible, most of the answers seemed legitimate. Besides focusing on the types of feedback students write, it could also be beneficial to inspect if the students have distinct profiles as assignment creators and reviewers. For example, are students who create highly graded or more complicated programming assignments also better reviewers in terms of insightful written feedback?

It would also be beneficial to dive deeper into the assignments students create, focusing especially on the students' background and experience. By analysing the highest and the lowest rated assignments and the types of assignments the students in these percentiles create, one could possibly find patterns, such as students with more programming experience creating more complicated assignments or just having higher completion rate in general.

6 Conclusion

In this study, a data set of 91 student-created programming assignments from the second week of introductory Java programming course was inspected to analyze the quality of both the assignments and their peer reviews. This is a continuation for a previous study [7] in order to analyze whether (novice) students can be reliably given grading tasks, in this case in order to automatize the screening process typical for crowdsourced content.

To summarize, the research questions and their answers are:

RQ1. What types of assignments do students create?
Answer: The student-created programming assignments are mostly short and simple, but adhere well to the instructions given.

RQ2. How do students perceive crowdsourced programming assignments?
Answer: The students tend to be more lenient in their reviews than the experts, and the grades they give have less deviation. This is in line with some previous studies, though the consensus regarding student perception for peer assessment remains inconclusive.

RQ3. Are experts and students in agreement with their reviews?
Answer: The agreement measured between experts and students was mediocre at best, and can only be used to draw tentative conclusions. When measuring reliability by statement, the noise caused by using all the peer reviewers as one reviewer lowers the agreement drastically (average $\alpha = 0.09$, slight agreement). When measuring

reliability by assignment, the results are more promising, though still lower end (average $\alpha = 0.27$, fair agreement).

RQ4. What characteristics are there in assignments in which experts and students agree the most?
Answer: The assignments with high alpha values are highly rated both by students and experts. They are relatively simple and similar to the programming assignments students are used to seeing on the course, but may contain novel topic ideas.

RQ5. What characteristics are there in assignments in which experts and students disagree the most?
Answer: The assignments with low alpha values tend to have lower ratings from both the students and experts, though the expert reviews use stricter grading. As noted in RQ2, students tend to grade using narrower scale, which explains the more drastic differences between the peer and the expert reviews.

In the future, the focus is on developing CrowdSorcerer for a more stable release with improved localisation and programming language support. Interesting future directions for potential studies focusing on both the assignments and the peer reviews are outlined in Section 5.3.

Though the current version of CrowdSorcerer has its limitations that, unfortunately, also reflect to the user experience and thus, possibly the assignment quality, the goal is that in the future, the tool could be used continuously for a longer period of time. This would allow more focused studies into the student-created assignment and peer review quality, and in time, using the tool to collect a databank of simple programming assignments with as little manual work from the educators as possible.

References

1 T. Volery and D. Lord, "Critical success factors in online education," *International Journal of Educational Management*, vol. 14, no. 5, pp. 216–223, 2000.

2 P. Weill and M. Broadbent, *Leveraging the New Infrastructure: How Market Leaders Capitalize on Information Technology.* USA: Harvard Business School Press, 1998.

3 J. Howe, "The rise of crowdsourcing," *Wired magazine*, vol. 14, no. 6, pp. 1–4, 2006.

4 L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum, "recaptcha: Human-based character recognition via web security measures," *Science*, vol. 321, no. 5895, pp. 1465–1468, 2008.

5 M. Dontcheva, R. R. Morris, J. R. Brandt, and E. M. Gerber, "Combining crowdsourcing and learning to improve engagement and performance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, (New York, NY, USA), p. 3379–3388, Association for Computing Machinery, 2014.

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 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 2018, (New York, NY, USA), ACM, 2018.

8 P. Denny, A. Luxton-Reilly, J. Hamer, and H. Purchase, "Coverage of course topics in a student generated mcq repository," in *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, (New York, NY, USA), pp. 11–15, ACM, 2009.

9 J. Hamer, H. C. Purchase, P. Denny, and A. Luxton-Reilly, "Quality of peer assessment in cs1," in *Proc. of the 5th International Workshop on Computing Education Research Workshop*, ICER '09, (New York, NY, USA), pp. 27–36, ACM, 2009.

10 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, (Darlinghurst, Australia, Australia), pp. 69–74, Australian Computer Society, Inc., 2008.

11 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.

12 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.

13 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.

14 N. Pirttinen and J. Leinonen, “Integrating crowdsorcerer: Lessons learned,” in *Proceedings of SPLICE 2019 workshop Computing Science Education Infrastructure* (P. Brusilovsky, T. Price, and S. Edwards, eds.), (United States), National Science Foundation (NSF), 8 2019.

15 E. Estellés-Arolas and F. González-Ladrón-De-Guevara, “Towards an integrated crowdsourcing definition,” *J. Inf. Sci.*, vol. 38, pp. 189–200, Apr. 2012.

16 M. Hossain and I. Kauranen, “Crowdsourcing: a comprehensive literature review,” *Strategic Outsourcing: An International Journal*, vol. 8, no. 1, pp. 2–22, 2015.

17 E. Schenk and C. Guittard, “Crowdsourcing: What can be outsourced to the crowd, and why?,” p. 29, 01 2009.

18 D. C. Brabham, “Crowdsourcing as a model for problem solving: An introduction and cases,” *Convergence*, vol. 14, no. 1, pp. 75–90, 2008.

19 “Amazon mechanical turk.” `https://www.mturk.com/`. Accessed: 25.6.2019.

20 K. Hara, A. Adams, K. Milland, S. Savage, C. Callison-Burch, and J. P. Bigham, “A data-driven analysis of workers’ earnings on amazon mechanical turk,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI ’18, (New York, NY, USA), pp. 449:1–449:14, ACM, 2018.

21 D. Difallah, E. Filatova, and P. Ipeirotis, “Demographics and dynamics of mechanical turk workers,” in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM ’18, (New York, NY, USA), pp. 135–143, ACM, 2018.

22 R. M. Ryan and E. L. Deci, “Intrinsic and extrinsic motivations: Classic definitions and new directions,” *Contemporary Educational Psychology*, vol. 25, no. 1, pp. 54 – 67, 2000.

23 Y. Wang and D. Fesenmaier, “Assessing motivation of contribution in online communities: An empirical investigation of an online travel community,” *Electronic Markets*, vol. 8, 01 1998.

24 A. C. Bandura and D. H. Schunk, “Cultivating competence, self-efficacy, and intrinsic interest through proximal self-motivation,” 1981.

25 R. Lukyanenko, J. Parsons, and Y. F. Wiersma, “The iq of the crowd: Understanding and improving information quality in structured user-generated content,” *Information Systems Research*, vol. 25, no. 4, pp. 669–689, 2014.

26 D. Schlagwein, D. Cecez-Kecmanovic, and B. Hanckel, “Ethical norms and issues in crowdsourcing practices: A habermasian analysis,” *Information Systems Journal*, 12 2018.

27 K. Fort, G. Adda, and K. B. Cohen, “Amazon mechanical turk: Gold mine or coal mine?,” *Computational Linguistics*, vol. 37, no. 2, pp. 413–420, 2011.

28 D. C. Brabham, “The myth of amateur crowds,” *Information, Communication & Society*, vol. 15, no. 3, pp. 394–410, 2012.

29 A. Settle, A. Vihavainen, and C. S. Miller, “Research directions for teaching programming online,” in *Proceedings of the 10th International Conference on Frontiers in Education: Computer Science and Computer Engineering*, 2014.

30 L. Pappano, “The year of the mooc,” *The New York Times*, 2012.

31 A. McAuley, B. Stewart, G. Siemens, and D. Cormier, *The MOOC model for digital practice.* University of Prince Edward Island, 2010.

32 A. Vihavainen, M. Luukkainen, and J. Kurhila, “Multi-faceted support for mooc in programming,” in *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE ’12, (New York, NY, USA), pp. 171–176, ACM, 2012.

33 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), pp. 177–182, ACM, 2013.

34 J. Leinonen, P. Ihantola, A. Leinonen, H. Nygren, J. Kurhila, M. Luukkainen, and A. Hellas, “Admitting students through an open online course in programming: A multi-year analysis of study success,” in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER ’19, (New York, NY, USA), p. 279–287, Association for Computing Machinery, 2019.

35 J. Kurhila and A. Vihavainen, “A purposeful mooc to alleviate insufficient cs education in finnish schools,” *Trans. Comput. Educ.*, vol. 15, pp. 10:1–10:18, Apr. 2015.

36 R. Vivian, K. Falkner, and N. Falkner, "Addressing the challenges of a new digital technologies curriculum: Moocs as a scalable solution for teacher professional development," *Research in Learning Technology*, vol. 22, Aug. 2014.

37 B. Ericson, M. Guzdial, B. Morrison, M. Parker, M. Moldavan, and L. Surasani, "An ebook for teachers learning cs principles," *ACM Inroads*, vol. 6, p. 84–86, Nov. 2015.

38 P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, (New York, NY, USA), pp. 86–93, ACM, 2010.

39 S. Fincher and M. Petre, *Computer Science Education Research.* 2004.

40 A. Leinonen, H. Nygren, N. Pirttinen, A. Hellas, and J. Leinonen, "Exploring the applicability of simple syntax writing practice for learning programming," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 84–90, Association for Computing Machinery, 2019.

41 D. Parsons and P. Haden, "Parson's programming puzzles: A fun and effective learning tool for first programming courses," in *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, (AUS), p. 157–163, Australian Computer Society, Inc., 2006.

42 P. Denny, A. Luxton-Reilly, and B. Simon, "Quality of student contributed questions using peerwise," in *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, (Darlinghurst, Australia, Australia), pp. 55–63, Australian Computer Society, Inc., 2009.

43 P. Denny, A. Luxton-Reilly, and J. Hamer, "Student use of the peerwise system," in *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, (New York, NY, USA), pp. 73–77, ACM, 2008.

44 P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, (New York, NY, USA), pp. 208–212, ACM, 2011.

45 A. Luxton-Reilly, P. Denny, B. Plimmer, and D. Bertinshaw, "Supporting student-generated free-response questions," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, (New York, NY, USA), pp. 153–157, ACM, 2011.

46 M. Barak and S. Rafaeli, "On-line question-posing and peer-assessment as means for web-based knowledge sharing in learning," *Int. J. Hum.-Comput. Stud.*, vol. 61, pp. 84–103, July 2004.

47 P. Denny, B. Hanks, and B. Simon, “Peerwise: Replication study of a student-collaborative self-testing web service in a u.s. setting,” in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE ’10, (New York, NY, USA), pp. 421–425, ACM, 2010.

48 K. Sanders, M. Ahmadzadeh, T. Clear, S. H. Edwards, M. Goldweber, C. Johnson, R. Lister, R. McCartney, E. Patitsas, and J. Spacco, “The canterbury questionbank: Building a repository of multiple-choice cs1 and cs2 questions,” in *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*, ITiCSE -WGR ’13, (New York, NY, USA), pp. 33–52, ACM, 2013.

49 N. Parlante, “Nifty reflections,” *SIGCSE Bull.*, vol. 39, pp. 25–26, June 2007.

50 C. A. Shaffer, V. Karavirta, A. Korhonen, and T. L. Naps, “Opendsa: Beginning a community active-ebook project,” in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling ’11, (New York, NY, USA), pp. 112–117, ACM, 2011.

51 P. Denny, A. Luxton-Reilly, and B. Simon, “Evaluating a new exam question: Parsons problems,” in *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER ’08, (New York, NY, USA), pp. 113–124, ACM, 2008.

52 Y. Hirai and A. Hazeyama, “A learning support system based on question-posing and its evaluation,” in *Fifth International Conference on Creating, Connecting and Collaborating through Computing (C5 ’07)*, pp. 178–184, Jan 2007.

53 A. Luxton-Reilly, D. Bertinshaw, P. Denny, B. Plimmer, and R. Sheehan, “The impact of question generation activities on performance,” in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE ’12, (New York, NY, USA), pp. 391–396, ACM, 2012.

54 A. Luxton-Reilly, P. Denny, B. Plimmer, and R. Sheehan, “Activities, affordances and attitude: How student-generated questions assist learning,” in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE ’12, (New York, NY, USA), pp. 4–9, ACM, 2012.

55 D. S. Weld, E. Adar, L. Chilton, R. Hoffmann, E. Horvitz, M. Koch, J. Landay, C. H. Lin, and Mausam, “Personalized online education"”a crowdsourcing challenge,” Association for the Advancement of Artificial Intelligence, January 2012.

56 J. Hui, A. Glenn, R. Jue, E. Gerber, and S. Dow, “Using anonymity and communal efforts to improve quality of crowdsourced feedback,” in *HCOMP*, 2015.

57 A. Korhonen, T. Naps, C. Boisvert, P. Crescenzi, V. Karavirta, L. Mannila, B. Miller, B. Morrison, S. H. Rodger, R. Ross, and C. A. Shaffer, “Requirements

and design strategies for open source interactive computer science ebooks," in *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*, ITiCSE -WGR '13, (New York, NY, USA), pp. 53–72, ACM, 2013.

58 F. J. Buckley and R. Poston, "Software quality assurance," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 36–41, Jan 1984.

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

60 D. Garvin, "What does "product quality" really mean?," *MIT Sloan Management Review*, vol. 26, pp. 25–43, 10 1984.

61 B. Edvardsson, "Service quality: beyond cognitive assessment," *Managing Service Quality: An International Journal*, vol. 15, no. 2, pp. 127–131, 2005.

62 B. Edvardsson, "Service quality improvement," *Managing Service Quality: An International Journal*, vol. 8, no. 2, pp. 142–149, 1998.

63 W. Abramowicz, R. Hofman, W. Suryn, and D. Zyskowski, "Square based web services quality model," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2008*, 03 2008.

64 F. T. Sheldon, K. M. Kavi, R. C. Tausworth, J. T. Yu, R. Brettschneider, and W. W. Everett, "Reliability measurement: From theory to practice," *IEEE Softw.*, vol. 9, pp. 13–20, July 1992.

65 T. Gilb, *Principles of Software Engineering Management.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

66 J. Tian, "Quality-evaluation models and measurements," *IEEE Software*, vol. 21, pp. 84–91, May 2004.

67 C. Prahalad and M. Krishnan, "The meaning of quality in the information age," *Harvard Business Review*, vol. 77, pp. 109–118, 9 1999.

68 "Software engineering - product quality - part 1: Quality model," Tech. Rep. ISO/IEC 9126-1:2001, International Organization for Standardization, 2001.

69 N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition.* Boca Raton, FL, USA: CRC Press, Inc., 3rd ed., 2014.

70 S. H. Kan, *Metrics and Models in Software Quality Engineering.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.

71 R. G. Dromey, "Cornering the chimera [software quality]," *IEEE Software*, vol. 13, pp. 33–43, Jan 1996.

72 B. Kitchenham, “Towards a constructive quality model. part 1: Software quality modelling, measurement and prediction,” *Software Engineering Journal*, vol. 2, pp. 105–126, July 1987.

73 B. W. Boehm, ed., *Characteristics of software quality.* North-Holland Pub. Co., 1978.

74 B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style.* New York, NY, USA: McGraw-Hill, Inc., 2nd ed., 1982.

75 D. M. Breuker, J. Derriks, and J. Brunekreef, “Measuring static quality of student code,” in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE ’11, (New York, NY, USA), p. 13–17, Association for Computing Machinery, 2011.

76 C. McDowell, L. Werner, H. Bullock, and J. Fernald, “The effects of pair-programming on performance in an introductory programming course,” *SIGCSE Bull.*, vol. 34, p. 38–42, Feb. 2002.

77 C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, “The impact of pair programming on student performance, perception and persistence,” in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 602–607, 2003.

78 B. Hanks, C. McDowell, D. Draper, and M. Krnjajic, “Program quality with pair programming in cs1,” in *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE ’04, (New York, NY, USA), p. 176–180, Association for Computing Machinery, 2004.

79 T. Brown, “Guided peer code reviews in cs1/cs2 and se curricula,” in *2019 IEEE Frontiers in Education Conference (FIE)*, pp. 1–2, 2019.

80 M. Stegeman, E. Barendsen, and S. Smetsers, “Designing a rubric for feedback on code quality in programming courses,” in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling ’16, (New York, NY, USA), p. 160–164, Association for Computing Machinery, 2016.

81 M. Stegeman, E. Barendsen, and S. Smetsers, “Towards an empirically validated model for assessment of code quality,” in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling ’14, (New York, NY, USA), p. 99–108, Association for Computing Machinery, 2014.

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

83 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.

84 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.

85 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), p. 907–913, Association for Computing Machinery, 2006.

86 C. Desai, D. S. Janzen, and J. Clements, “Implications of integrating test-driven development into cs1/cs2 curricula,” *SIGCSE Bull.*, vol. 41, p. 148–152, Mar. 2009.

87 D. Janzen and H. Saiedian, “Test-driven learning: Intrinsic integration of testing into the cs/se curriculum,” vol. 38, pp. 254–258, 03 2006.

88 J. Cohen, “A coefficient of agreement for nominal scales,” *Educational and Psychological Measurement*, p. 10, 04 1960.

89 W. A. Scott, “Reliability of content analysis: The case of nominal scale coding,” *The Public Opinion Quarterly*, vol. 19, no. 3, pp. 321–325, 1955.

90 J. L. Fleiss and J. Cohen, “The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability,” *Educational and Psychological Measurement*, vol. 33, no. 3, pp. 613–619, 1973.

91 K. Krippendorff, *Content Analysis: An Introduction to Its Methodology.* Sage Publications, Inc., 2004.

92 K. Krippendorff, “Estimating the reliability, systematic error and random error of interval data,” *Educational and Psychological Measurement*, vol. 30, no. 1, pp. 61–70, 1970.

93 K. Krippendorff and J. L. Fleiss, “Reliability of binary attribute data,” *Biometrics*, vol. 34, no. 1, pp. 142–144, 1978.

94 K. Krippendorff, “Recent developments in reliability analysis,” in *Proceedings of the 42nd Annual Meeting of the International Communication Association*, 05 1992.

95 J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

96 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), p. 117–122, Association for Computing Machinery, 2013.

97 J. Heywood, *Assessment in Higher Education: Student Learning, Teaching, Programmes and Institutions.* Jessica Kingsley Publishers, 2000.

98 L. A. Stefani, “Peer, self and tutor assessment: Relative reliabilities,” *Studies in Higher Education*, vol. 19, no. 1, pp. 69–75, 1994.

99 D. Boud, *Enhancing Learning Through Self-Assessment.* 01 1995.

100 D. Boud and N. Falchikov, “Quantitative studies of student self-assessment in higher education: A critical analysis of findings,” *Higher Education*, vol. 18, no. 5, pp. 529–549, 1989.

Appendix 1. Inter-rater reliability by assignment

Student 1	0.20	Student 30	0.05	Student 59	0.14
Student 2	-0.30	Student 31	0.32	Student 60	0.19
Student 3	0.02	Student 32	0.36	Student 61	0.65
Student 4	0.10	Student 33	-0.06	Student 62	0.31
Student 5	-0.35	Student 34	0.24	Student 63	0.66
Student 6	0.38	Student 35	-0.43	Student 64	0.51
Student 7	0.39	Student 36	0.29	Student 65	0.20
Student 8	0.82	Student 37	0.14	Student 66	0.96
Student 9	-0.24	Student 38	-0.30	Student 67	0.37
Student 10	0.15	Student 39	0.40	Student 68	0.49
Student 11	0.10	Student 40	0.73	Student 69	0.31
Student 12	0.39	Student 41	0.33	Student 70	0.69
Student 13	0.17	Student 42	0.57	Student 71	0.30
Student 14	-0.13	Student 43	0.54	Student 72	-0.03
Student 15	-0.26	Student 44	-0.24	Student 73	0.23
Student 16	0.19	Student 45	0.05	Student 74	0.44
Student 17	0.55	Student 46	0.39	Student 75	0.20
Student 18	0.00	Student 47	0.25	Student 76	0.57
Student 19	0.30	Student 48	0.47	Student 77	-0.63
Student 20	0.00	Student 49	0.95	Student 78	0.56
Student 21	0.41	Student 50	0.56	Student 79	0.32
Student 22	0.66	Student 51	0.19	Student 80	0.42
Student 23	0.39	Student 52	0.31	Student 81	0.54
Student 24	-0.07	Student 53	0.42	Student 82	0.51
Student 25	0.40	Student 54	-0.13	Student 83	0.09
Student 26	0.10	Student 55	-0.50	Student 84	0.58
Student 27	0.39	Student 56	0.55	Student 85	-0.35
Student 28	0.66	Student 57	0.57	Student 86	0.58
Student 29	0.63	Student 58	0.32	Student 87	0.38

Table 3: Ordinal Krippendorff's alpha for each student/assignment, the alpha values in the blue background.