IMPLEMENTATION OF AUTOMATED ASSESSMENT OF STUDENT ASSIGNMENTS ON AN INTRODUCTORY SOFTWARE TESTING COURSE

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering and Digital Transformation

2024

Lauri Pyykkö

Examiners: University Lecturer Erno Vanhala

Associate Professor Jussi Kasurinen

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Lauri Pyykkö

Implementation of automated assessment of student assignments on an introductory software testing course

Master's thesis

2024

61 pages, 12 figures, 4 tables and 2 appendices

Examiners: University Lecturer Erno Vanhala

Associate Professor Jussi Kasurinen

Keywords: Automated assessment, Software testing, CodeGrade, Programming education

The role of automated assessment in programming education has been gaining popularity in recent years as the course participant numbers have continued to increase. This thesis details the process of introducing an automated assessment system, CodeGrade, on an introductory level software testing course with the goal of updating existing programming assignments and moving towards completely automatic assessment process. The purpose of this paper is to find out how CodeGrade performs in assessing student submissions and what methods can be used to successfully evaluate students' testing skills. A literature review is conducted to find out recent themes in automated assessment as well as best practices in implementing programming assignments and their assessment process. Updated programming assignments and the performance of CodeGrade were evaluated through a course-end survey on a Likert-scale of 1–5, resulting in a positive reception with CodeGrade having received an average grade of 4.2 and course assignments an average grade of 4.3. Results showed that there was still room for improvement, especially in terms of introducing programming assignments that handled topics beyond unit testing. Additionally, the assessment process could be further improved with a combination of tools to improve accuracy and provide quality feedback even in cases where the assessment process is unable to be completed.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT School of Engineering Science

Tietotekniikka

Lauri Pyykkö

Opiskelijatehtävien automaattisen arvioinnin toteutus ohjelmointitestauksen johdantokurssille

Diplomityö

2024

61 sivua, 12 kuvaa, 4 taulukkoa ja 2 liitettä

Tarkastajat: Yliopisto-opettaja Erno Vanhala

Apulaisprofessori Jussi Kasurinen

Avainsanat: Automaattinen arviointi, Ohjelmistotestaus, CodeGrade, Ohjelmoinnin opetus

Automaattisen arvioinnin rooli ohjelmoinnin opetuksessa on kasvattanut suosiotaan viime vuosien aikana kurssin opiskelijamäärien jatkaessa kasvuaan. Tämä opinnäytetyö kuvaa automaattisen arviointijärjestelmän, CodeGraden, käyttöönottoprosessia ohjelmointitestauksen johdantokurssilla, tarkoituksena päivittää olemassa olevat ohjelmointitehtävät ja siirtyä kohti täysin automaattista arviointiprosessia. Tämän työn tarkoituksena on selvittää, miten CodeGrade suoriutuu opiskelijoiden testaustaitojen arvioinnissa ja mitä metodeja voidaan käyttää opiskelijoiden testaustaitojen arviointiin. Kirjallisuuskatsaus toteutetaan selvittämään automaattisen arvioinnin viimeaikaisia teemoja sekä parhaiksi todettuja käytäntöjä ohjelmointitehtävien toteutusta ja niiden arviointia varten. Päivitettyjä ohjelmointitehtäviä ja CodeGraden toimintakykyä arvioitiin Likertin-asteikolla arvosanoin 1–5. Tuloksena oli positiivinen vastaanotto: CodeGrade arvioitiin keskimääräisellä arvosanalla 4.2 ja kurssin tehtävät keskimääräisellä arvosanalla 4.3. Tulokset osoittivat, että parantamisen varaa on vielä etenkin uusien ohjelmointitehtävien osalta, jotka käsittelisivät muita testauksen aihealueita yksikkötestauksen lisäksi. Lisäksi arviointiprosessia voidaan jatkokehittää yhdistelemällä erilaisia työkaluja arvioinnin tarkkuuden ja annetun palautteen laadun lisäämiseksi etenkin tilanteissa, joissa arviointiprosessia ei pystytä suorittamaan loppuun onnistuneesti.

SYMBOLS AND ABBREVIATIONS

AA	Automated Assessment
AAT	Automated Assessment Tool
CG	CodeGrade
CPU	Central Processing Unit
DSR	Design Science Research
LMS	Learning Management System
LTS	Long Term Support
LUT	Lappeenranta-Lahti University of Technology
MC/DC	Modified Condition/Decision Coverage
OOP	Object-Oriented Programming
SAT	Static Analysis Tool
TA	Teaching Assistant
TDD	Test-Driven Development
VM	Virtual Machine

Table of contents

Appendices

1 Introduction

Automated code assessment has been around for long time. In his paper Hollingsworth (1960) details a positive experience of using automatic graders in being able to speed up grading and thus generating savings when teaching programming to a large number of students. Since then, automatic code assessment has gone through variety of changes, which has resulted in even more possibilities utilizing automated assessment tools (AAT) as part of programming education. AATs provide many benefits to programming education such as objectivity, consistency, speed and high availability for both assignments and given feedback (Ala-Mutka, 2005). Especially students taking introductory programming courses might not be familiar with testing their own code and tend to only focus on correctness of program output instead of actual functionality of the code (Aaltonen et al., 2010). With the capabilities and benefits provided by different AATs, student testing skills can be evaluated while also providing students with valuable feedback to facilitate effective learning.

In the span of one year between 2022 and 2023 the student numbers on an introductory LUT University ran software testing course *Fundamentals of Software Testing* had increased from 142 to 168 which equates to roughly 18% increase in participation. Ultimately, the rising student numbers meant that there was a need to introduce tools to ease the grading process while being able to provide feedback to the students on some of the general issues they are having. Without a tool to support automated grading the teaching assistants (TA) would effectively have to evaluate each programming assignment using traditional code inspection methods and coordinating with each other to ensure that the grading is as fair as possible. Additionally, with student numbers almost reaching close to 200 participants, manually grading two essay-type assignments and a programming task each week required significant effort from the TAs.

Together with assignment problem definitions leaving some room for interpretation and lack of formal grading guidelines there was a clear benefit in introducing automated assessment on the introductory software testing course. CodeGrade is an automated assessment tool that was initially selected to be used by the software engineering department as a result of a selection process conducted in LUT University during the year 2021. The main selling points of CodeGrade were named to be its usability, range of features and compatibility with the existing Learning Management System (LMS), Moodle (Mäkelä, 2021). CodeGrade had been previously piloted on some of the early software engineering courses but had still not been introduced in the context of software testing education.

1.1 Research questions and limitations

The goal of this thesis is to explore different approaches to automated assessment of programming assignments in the context of LUT University ran course *Fundamentals of Software Testing*. The course is conducted using Java as a primary programming language with the course having prerequisites in object-oriented programming through an introductory Java-course. As a result, the course assignments and their automated assessment are limited to testing frameworks and analysis tools with support for Java programming language. Automatic assessment of student solutions is conducted using CodeGrade, which works as a further limiting factor when considering different methods in assessing student testing skills.

This thesis aims to answer the following research questions:

1. How does Codegrade perform in automatic assessment of student written tests?
    a. How well does CodeGrade facilitate learning and how easy is it to use from students' perspective?
    b. How well does CodeGrade perform when assessing student solutions?
2. What methods can be used to automatically assess the student's testing skills?
    a. How to evaluate correctness of student solutions?
    b. What techniques and tools can be used?
3. How should the course assignments be adapted to CodeGrade?
    a. What changes should be made to course assignments to meet the needs of the students and allow easy adaptation to CodeGrade and to promote learning?

The first research question has to do with evaluation of the AAT in the context of software testing skills from students' perspective. It is mostly answered by conducting a survey during the Fundamentals of Software Testing course, presenting the data and discussing the results as a part of this thesis. The second and third research questions are answered initially through a literature review and during the implementation of course assignments as the limitations of used automated assessment system become more apparent. The answers to the third research question are discussed during the final parts of this thesis, after first having presented the results.

1.2 Thesis structure

This thesis consists of eight primary chapters: Introduction, Literature review, Research methods, Previous course structure, Implementation, Results, Discussion, Conclusion. The first chapter aims to give a brief introduction to the thesis topic by providing background information, brief problem statement, goals, and limitations. The second chapter summarizes previous findings of relevant research and attempts to illuminate some of the central testing related concepts. The chapter starts by outlining different automated assessment techniques, their benefits and potential pitfalls, while focusing on their applications in an educational setting, especially their applications in programming education. The third chapter presents the research methodology used to conduct this thesis, after which in chapter four the previous structure of the course *Fundamentals of Software Testing* is presented as background information leading up to chapter five. In this chapter the implementation of the updated course and the automated assessment process is presented. The sixth chapter presents results obtained from the conducted survey and the data collected from the course Moodle pages. In the seventh chapter the results are discussed. Finally, in the eighth chapter summary of this thesis is presented and implications for the future are discussed.

2 Literature review

This chapter examines previous literature on automated assessment in the context of programming education. The literature review starts with short summary of advantages and best practices in implementing automated assessment systems, after which some of the related challenges are discussed. Following this, a categorization for AA systems is presented and different methods for assessing programming assignments are described in a broad setting. Finally, the importance of teaching testing skills and the role of test-driven development in education is discussed.

Search for literature was carried out using Google Scholar and LUT Primo as the primary databases. Google Scholar was used as the initial starting point using keywords related to automated assessment such as: “*automated assessment education*” and “*automated assessment programming*”. LUT Primo was also used to find literature but was typically used in cases where literature on more specific topics was needed or a publication was not available through Google Scholar. The initial search using the mentioned keywords worked to limit papers to a selection of topics having to do with programming assignments and automated assessment systems used in education. Literature was then selected with the publication date and number of citations as the biggest contributors in finding some of the more esteemed and recent publications. Abstracts were used as a primary identifier whether the study would be included or not. However, in cases where the quality of the abstract was low or the abstract did not contain enough information to decisively rule it out, conclusions were also reviewed to finally create a starting set for the selection process.

The selection of literature was conducted using a snowballing approach. Snowballing refers to the process of using the reference list or citations of a study to identify additional relevant studies in an iterative fashion (Wohlin, 2014). Snowballing can be conducted as either forward facing or backward facing process depending on whether the list of references or citations in other papers are used. Instead of conducting both forward and backward snowballing as suggested by Webster and Watson (2002), backwards snowballing was deemed to provide an exhaustive collection of literature for this thesis.

2.1 Automated assessment best practices

As the number of enrolments on a programming course increases, the workload course personnel are subjected to affects the overall quality of grading and as a result also impacts students' academic performance (Avramov, 2022; Galan et al., 2019). AATs are able to reduce workload by taking over some of the more time-consuming parts of code assessment such as assessing correct program behaviour, code quality and even providing valuable feedback that expedites learning (Ala-Mutka, 2005). In their research article Zampirolli et al. (2021) discuss some the experiences regarding their implementation of automated assessment system on an introductory programming course that has enrolled thousands of students annually. Some of the mentioned notable benefits include faster feedback and uniform assessment of exercises. Uniform assessment can be seen especially important on courses that enrol a large number of students as often grading without automated assessment (AA) involves subjective opinion of what constitutes a quality program, even when using uniform grading criteria. Additionally, the complexity of the program creates further challenges for the instructor to accurately assess the program simply by reading the source code and plagiarism detection on these large online courses is often very inefficient or impossible just by relying on human graders. (Avramov, 2022; Cheang et al., 2003; Kay et al., 1994; Zampirolli et al., 2021)

Many success factors have been identified in related literature. Pieterse (2013) discusses some of the more integral success factors and identifies them as: *quality assignments*, *clear formulation of tasks*, *well-chosen test data*, *quality feedback*, *unlimited submissions*, *student testing maturity* and *additional support*. One notable benefit of AA systems, that was already briefly discussed, is their ability to free course instructors and assistants from having to manually go through every submission. The need for *quality assignments and clear task formulation* comes from the inherent differences of AA compared to manual grading. Unlike with manual grading, which includes human interaction, AA systems need clearly established grading criteria and are not able to account for solution creativity in the same way a human could (Pieterse, 2013). This in turn creates the need for uniform assessment process so that student performance can be equally measured (Zampirolli et al., 2021).

Many experiences on large scale online courses highlight the importance of allowing *unlimited submissions*, reasoning that immediate feedback given by AA systems provides instant gratification and allows students to correct errors in real time without having to wait for feedback from course personnel (Edwards, 2014). This in turn results in motivating the students and reinforcing iterative learning. Furthermore, some of the more well known problems with manual grading is its difficulty in providing consistent, timely and *quality feedback* (Mekterović et al., 2020). In their research to different features of learning management systems (LMS) Rößling et al. (2008) present the ability to provide immediate quality feedback as one of the most important and requested features, especially in smaller scale LMSs such as tools intended for code evaluation. However, not all experiences with immediate feedback are positive; in their publication Kyrlilov and Noelle (2015) detail their experience with using an automated assessment system to provide instant feedback on programming assignments and report that students receiving binary feedback, i.e. “correct” or “incorrect”, were twice as likely to cheat. This further emphasizes the importance in providing quality feedback that is not limited to simple binary feedback. (Ala-Mutka, 2005; Malmi et al., 2002; Pieterse, 2013)

2.2 Automated assessment challenges

Submitted program may include flaws or malicious code that can cause damage to the running environment by accessing restricted areas, causing server downtime or pose a significant security risk otherwise (Ala-Mutka, 2005; Forišek, 2006; Rahman and Nordin, 2007). This problem is relatively easily managed with the introduction of a sandbox that executes code in a secure isolated environment. However, even with an isolated sandbox environment these threats do not completely disappear. Forišek (2006) details some of the attacks a sandbox environment can be subjected to during computer science contests, such as attacks that block execution or consume excessive amount of system resources. This can be easily expanded to include grading systems used in programming education as the environment in both is structured for similar use cases. It is also good to note that not all attack types require malicious intent from the user but can also be caused by simple design flaws in their own code. AA systems are often integrated with different LMSs which may also become a target for malicious attacks. Integrating with LMS is beneficial but it is good to be aware that the AA system is not the only critical point of entry for malicious attacks. One of the challenges for LMSs is the authentication of users in a secure manner. Identity federation of the Finnish universities, HAKA, is an example of a project that aims to answer this challenge by developing and maintaining infrastructure for secure authentication (Rößling et al., 2008). (Ala-Mutka, 2005; Forišek, 2006; Ihantola et al., 2010)

When receiving feedback through AAT, especially with unlimited submissions, the students have a tendency to focus on passing all the checks and achieving passing mark without necessarily considering other more fundamental issues with their code such as design issues (Chen, 2004; Edwards, 2014; Pieterse, 2013). Main consideration for the student is to pass each test without necessarily understanding what is being tested and how. This way, introduction of automated assessment systems on programming courses can inevitably result in diminished learning when it comes to student's own testing skills (Chen, 2004). While this might seem like a significant drawback to using AA systems, this can be addressed by adapting the system to assess student submitted test cases alongside the actual program code, thus ensuring the students' testing skills don't fall behind. It is a different matter altogether on how this is achieved and how effectively test cases are assessed. However, several university driven projects have reported positive outcomes for this approach (Chen, 2004; Edwards, 2014; Jackson and Usher, 1997). Another solution is simply limiting the number of submissions allowed for students or by introducing a wait time between submissions, although these limiting factors have been perceived to negatively affect learning and can be seen as an obstacle, especially with programming education types using test-driven development (TDD) (Edwards, 2003a). It is also possible to limit feedback given to the student after submitting their code. However, this can cause confusion especially among students that are distrusting towards AA systems, causing the students to make wrong judgements on what the actual issue is. TDD is discussed further in section *2.5 Test-driven development and assessment of testing skills*.

Another challenge for AA systems is the issue of balancing quality with speed and availability. One of the biggest benefits of automated assessment is the speed at which code can be evaluated compared with needing the course instructor or assistant to manually go through each assignment. This also often results in diminished feedback quality and consequently the benefit gained from speed and availability varies based on the number of course participants and the complexity of assignments. In the case of Massive Open Online Courses (MOOCs) that typically have little to no entry requirements the importance of both speed and availability of assessment is emphasized due to the large number of participants (Pieterse, 2013). However, with complex assignments it is often more beneficial to use a semiautomatic approach instead of a fully automatic approach, whereas a simple assignment is easy to tailor in a way that supports fully automatic assessment. (Ala-Mutka, 2005)

2.3 Automated assessment categorization

One of the more common ways to categorize AA systems and tools is by division into three distinct categories based on how they assess the program code*: Dynamic*-, *Static*- and *Hybrid* assessment tools (Ala-Mutka, 2005; Arifi et al., 2015; Galan et al., 2019). Hybrid assessment simply combines features from both static and dynamic assessment methods which is typical for most assessment tools as by themselves dynamic- and static methods each have some inherent weaknesses (Galan et al., 2019). Following sections discuss dynamic- and static assessment methods and some of their weaknesses as described by related literature. Different features that can be assessed with each assessment type are described in Table 1.

2.3.1 Dynamic assessment

Dynamic assessment is the most popular assessment type (Galan et al., 2019). Dynamic assessment methods evaluate the target program by executing the program against a test case with the intention of covering different execution paths and creating different states in the program (Figure 1). The output of the program is then compared to a model solution to ascertain correctness of the program. This general description of dynamic assessment assumes that the program can be executed, however in presence of coding errors this may not always be possible. One possible solution to this problem is evaluating the program in smaller pieces, i.e. separately assessing each function and grading the program as a sum of its functions. (Arifi et al., 2015)

One of the bigger weaknesses of dynamic assessments is its inability to give meaningful feedback in cases where the program does not compile and consequently the student is only shown an error message of failed program compilation. Additionally, even in cases where the program compiles and finishes executing successfully, the feedback may be on a very general level and at the mercy of test-case completeness as typically failed test-cases tend to provide more detailed feedback on potential problems.

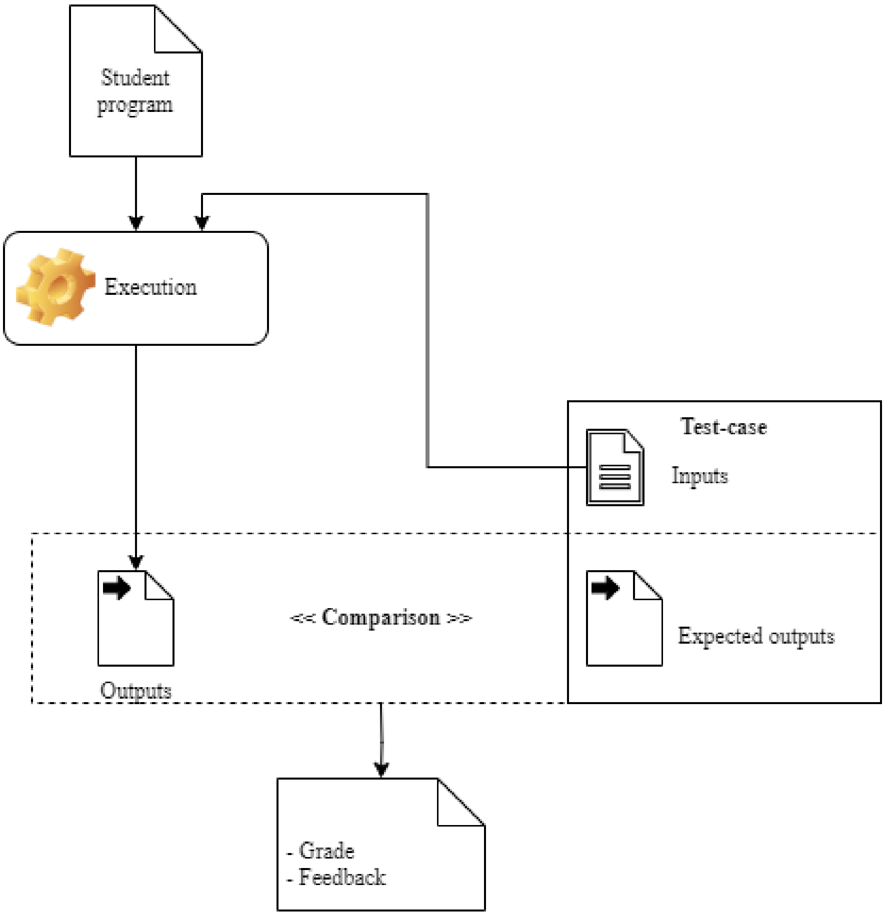

Figure 1. Simplified dynamic assessment method (Arifi et al., 2015).

2.3.2 Static assessment

Static assessment tools (SATs) and methods are used to detect functional problems such as faults, vulnerabilities and code smells and can also be used to address maintainability related problems such as conforming to certain language specific coding standards (Beller et al., 2016; Zampetti et al., 2017). Static code analysis is often used in conjunction with dynamic analysis tools and methods, but static analysis still has clear benefits in being able to find issues with the code that dynamic methods may have missed. Static analysis is especially useful in situations where code can't be executed dynamically as SATs are able to evaluate the program without actually having to execute it (Ala-Mutka, 2005; Lenarduzzi et al., 2023).

Coding style is a common static assessment target as coding style directly relates to understanding and thus maintainability of the code (Ala-Mutka, 2005). *Checkstyle* and *PMD* are examples of static code analysis tools that can be used to assess coding quality. Checkstyle is used to evaluate Java coding style to ensure the code conforms to a determined style configuration guide such as Google Java Style or Sun Java Style (Lenarduzzi et al., 2023). Checkstyle inspects the code against a long list of different rules that are further categorized into: annotations, block checks, class design, coding, headers, imports, Javadoc comments, metric, miscellaneous, modifiers, naming conventions, regexp, size violations and whitespace. Similarly, PMD also works based on different rules that are grouped under following categories: code style, design, documentation, error prone, multithreading, performance and security (Checkstyle, 2024; PMD Source Code Analyzer Project, 2024; Lenarduzzi et al., 2023).

A persistent problem with static code assessment tools is the number of false positives, namely the amount of false warnings caused by used rules (Lenarduzzi et al., 2023; Thung et al., 2015). Static code analysis tools have been observed to have varied performance in detecting faults depending on the target project. In their research Thung et al. compare five different static analysis tools: *FindBugs*, *JLint*, *PMD*, *CheckStyle*, *JCSC* and detail how these tools succeeded in finding faults on different open-source projects. The results indicate that performance of static analysis tools varied noticeably depending on the type of project they were used to inspect. Additionally, the faulty lines as marked by the tools were often indicated very broadly and accompanied with generic warning messages. Consequently, the developer would have to be very sceptical whether an actual fault was detected and if the indicated fault is directly related to given warning message. Confirming each warning to be authentic instead of a false positive is time consuming especially in larger projects and according to some estimates takes three to eight minutes per warning (Beller et al., 2016, p. 2).

Regardless, the importance of SATs should not be overlooked as a part of quality assurance activities, especially in the earlier stages of software development. In their research Zheng et al. (2006) examine the economic costs of using SAT and evaluate the effectiveness compared to other static analysis techniques such as code inspection. The results of their research indicate that the costs related to SAT and actual fault detection yields are similar in scale when compared to traditional inspections. However, In the context of programming education, the value of these tools is further emphasised as means to provide valuable feedback on potential problems due to lack of experience from students which directly affects code inspection quality.

Table 1. Categorization of automated assessment systems and features assessed by them.

Assessment approach	**Features assessed**	**Reference**
Dynamic	**Functionality**: Running a program against a test-case and comparing output or return values to intended values. One of the main features of most automated assessment tools.	(Ala-Mutka, 2005; Edwards, 2014)
	Efficiency: For example, measuring the time it takes for a program to terminate or to accomplish a certain functionality.	
	Testing skills: Many tools such as Assyst and Web-CAT are able to assess student test-suites along with submitted program code. For example, by requiring students to submit test suites along with the program code.	
	Resource usage: It is also possible to assess resource usage during program execution, such as correct memory management.	
Static	**Style**: Readability of the program. Covers aspects such as variable naming, use of constants, line spacing, commenting, program layout.	(Arifi et al., 2015; Ihantola et al., 2010; Rahman and Nordin, 2007)
	Errors: Syntax and semantic errors in program code. *Syntax errors* refer to the use of incorrect grammar, i.e. syntax of the programming language. This includes errors such as typos and missing semicolons. Syntax errors are	

	typically easy to detect as they are detected by compilers. *Semantic errors* are logical flaws in the code that are common especially with new programmers. Semantic errors can be hard to spot since they are typically missed during program compilation.
	Metrics: measurable parts of the program such as complexity, quality or performance. Calculates numerical representation of properties such as: number of statements, frequency of comments, cyclomatic complexity.
	Key-word matching: Looking for certain pre-determined keywords from program code. Can be useful for structural checks or confirming certain functions were or were not used as instructed.
	Similarity: Structural and non-structural similarity. Can be used to assess student solutions against answer scheme provided by instructor or similarity in terms of plagiarism detection. Suitable only for small and simple programs.

2.4 Automated assessment tools in education

Majority of AA systems developed for programming education purposes have built in support for Java or include Java as one of the supported languages. The decision to support Java as one of the languages can be explained by its popularity as one of the introductory programming languages. Many examples of different AA systems exist in related literature that are typically intended for a specific use case such as assessing programming exercises and providing feedback, or they can even be more specialized systems such as systems for plagiarism detection. These tools are often created as part of a thesis or to be used as a part of a certain university course and are rarely used outside the intended scope. They often remain in use only within the original institution regardless of how useful they are perceived to be. This section introduces some of the more well-known automated assessment tools that have been previously used in programming education. (Ihantola et al., 2010; Rößling et al., 2008)

Web-Cat is an automated testing tool created by Virginia Tech that is designed to be integrated with learning management systems (LMS). As of the end of year 2020, Web-CAT had been used across 39 different universities and today the number is likely to be even higher (Web-CAT, 2019). Web-CAT uses a TDD-oriented approach that requires submission of test suite along with student's own program code. The core ideology for the tool is to allow students to demonstrate the correctness of their code themselves instead of having to rely on indicators of correctness that are simple results of comparing their own program output to example solutions. Grading of assignments is done by giving students different scoring corresponding a certain code quality metric such as validity, completeness and correctness. In practice, Web-CAT runs student test suite against example solution to determine the validity of each test case. In addition to comparing program output, scoring also considers code coverage achieved by student submitted tests. (Edwards, 2014; Virginia Tech, 2019)

Assyst, developed in the University of Liverpool, is one of the earlier implementations of automated assessment tools. The tool allows management of course assignment related grading operations such as handling student submissions, organizing assignments, grading submissions and sending reports back to students. Correctness of student solutions is evaluated with simple I/O-tests where program output with certain input is compared to example output as specified by instructor's model solution. Other used metrics include efficiency, complexity, style and test data adequacy. (Jackson and Usher, 1997)

In the case of Assyst, *efficiency* of the program is the measure of central processing unit (CPU) time spent executing the program (Jackson and Usher, 1997). *Complexity* is determined using McCabe's metric for complexity, otherwise known as cyclomatic complexity, i.e. measure of different paths through the program code (McCabe, 1976, p. 1). *Style* of the program is evaluated through simplified indicators such as module and identifier length, comment frequency, number of indentations and blank lines (Berry and Meekings, 1985). Finally, *Test data adequacy* corresponds to statement coverage on code execution, i.e. the number of executable statements covered by student's own test data (Jackson and Usher, 1997).

2.5 Test-driven development and assessment of testing skills

TDD (Test-driven development) follows a process where the developer starts by writing unit tests for the code they are about to implement and this is then repeated continuously as functionality is ready to be implemented (Maximilien and Williams, 2003). The assumption is that implemented code should always be able to pass the created unit tests. On the contrary, this also means that code is being written as a response to failing test cases. As a result, TDD promotes incremental development practices and allows developers to feel more confident about their own code functioning as it was designed. Granted, this is all dependent on the correctness of their test case design. (Edwards, 2003)

As was mentioned previously when discussing some of the challenges that come with automated assessment, it is typical for students to overly focus on the most immediate feedback which, in most cases, tends to be the output correctness. For example, in a case where the program does not compile, the assessment tool would grade the program with a zero and the student would not pay attention to anything past that. As a result, other important aspects such as design and testing skills are put aside. Teaching testing skills is important for several reasons; when the student is encouraged to perform testing activities as a part of writing their program code, they also get to learn the flow of their program more effectively (de Souza et al., 2014). Another more indirect example is a commonly heard phrase in software engineering according to which the longer a defect remains in the software, the more difficult and costly it becomes to remove (Maximilien and Williams, 2003). This might not have immediate consequences for the students but is still important to emphasize early on. TDD attempts to solve this issue by helping to detect and remove possible defects before they become costly. Additional benefits include being able to teach students how to produce code that is automatically verifiable, thereby enhancing code reliability, and also creating a test bed for regression testing (Maximilien and Williams, 2003). (Edwards, 2014; Edwards, 2003)

In the previous section, two well-known automated assessment tools that both include functionality for assessing students' tests were discussed. In fact, most of the existing AA tools use code coverage criteria such as statement coverage, branch coverage, modified condition/decision coverage (MC/DC) as a test adequacy criteria (Hemmati, 2015; Ivankovic and Petrovic, 2019). In simple terms, code coverage is the measure of parts of the program that did not execute during one or more runs of the program (Tikir and Hollingsworth, 2002). *Statement coverage* measures the number of executable statements covered by the test suite. However, it is seen as one of the weaker coverage indicators due to it being easy to satisfy but still remains one of the most used coverage measures (Hemmati, 2015; Inozemtseva and Holmes, 2014). *Branch coverage* measures the number of different branches covered, i.e. paths that are taken by the test suite and is seen as a stronger form of a test adequacy indicator. *MC/DC* breaks down each decision point in code to smaller atomic conditions and evaluates them separately to form a compound decision. While being more complex, it is also the strongest test adequacy indicator out of the discussed coverage criteria. (Hemmati, 2015)

Despite being an effective and commonly used tool, code coverage doesn't always guarantee test adequacy even when combined with other static analysis methods (Hemmati, 2015). In their paper, Aaltonen et al. propose mutation analysis as an alternative to code coverage methods. One of the primary problems with code coverage was seen to be its possibly deceptive results even with poor test quality. For example, students are able to write unit tests that do not correctly assess the functionality of the target method but still achieve high coverage simply because a method was called, or different paths were covered. Mutation analysis is a technique used to evaluate existing tests by introducing new artificial defects to the program code that is being tested. Mutants, i.e. combinations of errors are generated in large quantities and subsequently the tests are evaluated against created mutants. Tests that are able to distinguish mutants from the actual program can be seen to have passed the test objectives (Papadakis et al., 2019). Consequently, if a mutant is distinguished it is then flagged as "killed". The objective of mutation testing is killing the mutants and to this end a mutation score is calculated. Mutation score serves as an indicator of how many mutants were killed compared to how many killable mutants were missed by designed test cases. The term "killable mutants" is used to indicate that not all mutants can be killed as some mutants are seen as being equivalent to the original program and thus creating inaccuracies when calculating the mutation score. Another source for inaccuracies is redundant mutants, i.e. mutants that are killed as a result of another mutant being killed. This way the redundant mutants do not contribute to the process but still inflate the mutation score. (Kintis et al., 2018; Papadakis et al., 2019)

3 Research methods

This thesis is conducted using the Design Science Research (DSR) methodology. The general outline of the DSR process in information science consists of repeated build and evaluate loops where knowledge is drawn from existing knowledge base and used to produce an artifact which is then evaluated and gained knowledge added to the knowledge base (Brocke et al., 2020; Hevner et al., 2004). In the case of this thesis, the iterative nature is partly limited due to time constraints created by the gap between different course iterations. Instead, the artifact can be further improved by using this research as an initial starting point.

The activities included in the DSR process vary based on definition and may be split into several subcategories. This thesis uses the definition presented by Peffers et al. (2020), according which DSR process is divided into following activities: *problem identification and motivation*, *objectives of a solution*, *design and development*, *demonstration*, *evaluation and communication*.

Problem identification and motivation phase starts with defining the research problem and consequently reviewing existing research to deepen understanding of the problem and its possible solutions. Initial identification of the problem is drawn from the existing environment which includes people, organizations and current or planned technologies (Brocke et al., 2020). For this thesis, the initial problem was identified when the increasing number of course participants in the Fundamentals of Software Testing -course resulted in teaching assistants (TA) having to manually grade hundreds of assignments. Multiple TAs were needed to grade each assignment which in turn could lead to variance in grading between similar submissions.

Objectives of a solution are defined as the need to re-evaluate the course assignments and to produce automated assessment solutions that are capable of effectively evaluating students' testing skills while providing valuable feedback. After the objectives have been defined, an artifact is created during the *design and development* -activity. The design and development of the artifact is grounded in knowledge gained during the literature review while also somewhat restricted by capabilities offered by CodeGrade. *Demonstration* and *evaluation* are conducted by implementing the produced artifact on the next iteration of the course and subsequently receiving feedback from the students in the form of a course-end survey where the students are asked to evaluate different aspects of the course, including their experience with programming assignments and the automated assessment system. The results are then communicated in the latter part of this paper under chapter 6 *Results*.

4 Previous course structure

The purpose of this section is to describe the structure and demographic of the course *Fundamentals of Software Testing* during the fall 2022 iteration of the course, i.e. before the introduction of AAT. Fundamentals of software testing is a second-year course ran by the computer sciences department with object-oriented programming (OOP) course as a compulsory prerequisite and some recommended introductory programming course prerequisites. OOP is being taught in Java and therefore students enrolled on the course are expected to have prior knowledge of programming in Java.

The course was split into seven weeks of topics and their corresponding assignments (Figure 2), with last week being omitted and replaced by a guest lecture. The weekly assignments are a fairly even mix of either writing assignments or programming assignments. The programming assignments typically required the students to submit their solution in a ZIP-format. There were no further requirements on the structure of the submission, which typically resulted in TAs having to solve some peculiarities when grading programming assignments.

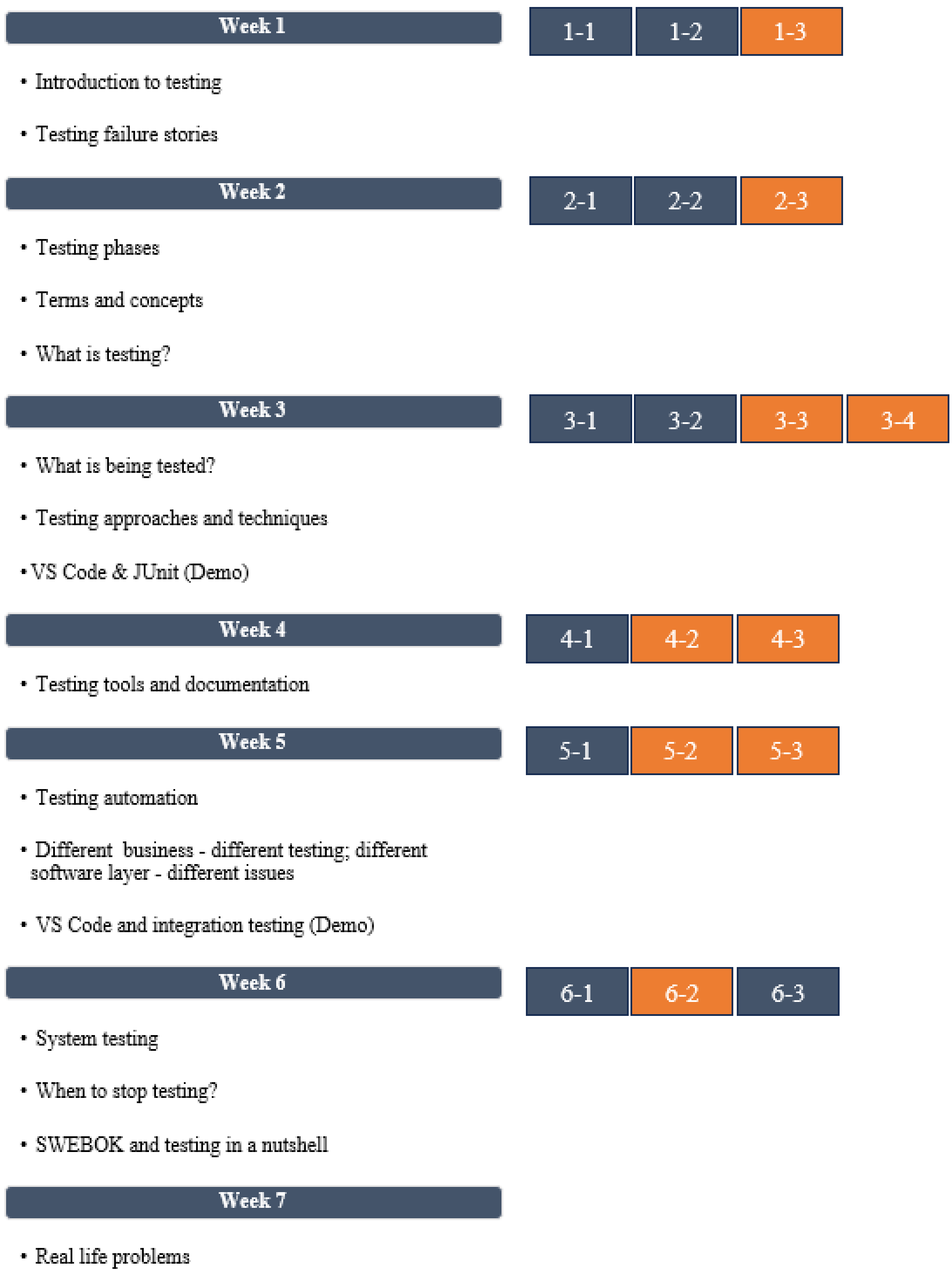

Figure 2. Lecture contents of each week and their corresponding weekly assignments. Practical assignments that require coding are marked in orange.

The course grading was split into several different categories (Table 2) out of which weekly assignments were optional so long as the student was able to reach their point target with other compulsory or non-compulsory activities. To pass the course the student was required to achieve a total of 50 points, which equates to half of the total points available. Both the course exam and project are worth 30 points each which also makes it possible to pass the course without weekly assignments. However, students were still required to obtain at least half of the total points available from each of the compulsory activities. This means that a student could not, for example, pass the course without achieving 15 points from the course project.

Table 2. Course grading categories and available points during year 2022.

Category	**Available points**	**Compulsory**
Exam	30	Yes
Project	30	Yes
Weekly assignments	30	No
Guest lecture	10	No

5 Implementation

The purpose of this chapter is to detail the implementation of the updated version of a software testing focused course *Fundamentals of Software Testing*. This chapter starts with a description of the new course structure and then moves on to detail different tools that make up the course environment. Finally, course programming assignments and their assessment process are described.

5.1 Updated course structure

The basic structure of the course remained the same with a total of seven weeks of lectures and related assignments. Lecture topics remained mostly the same with the exception of practical tutorial videos moved up to first week in order for students to be better equipped for the programming assignments. The biggest difference to the previous course implementation is the change in weekly assignment structure which was updated to a more consistent format of three weekly assignments: (1) Quiz, (2) Programming assignment, and (3) Writing assignment. *Quizzes* require students to answer some basic questions on corresponding weeks lecture contents and are mostly a tool to ensure lecture topics were learned. They also work as a starting point for each weekly assignments as the other assignments get unlocked only after completing that weeks' Quiz-activity. *Programming assignments* are assignments where students are asked to solve a testing related task by writing actual program code. Programming assignments are all integrated with CodeGrade and consequently require no manual grading from course assistants. *Writing assignments* are mostly free-form submissions that answer a more in-depth question and vary from typical essay-type answers to conducting black-box testing and documenting planned test cases. While the writing assignments varied in content, this division of assignments is reasoned by other assignments not requiring writing code or interaction with the CodeGrade system. Another difference to the previous course is the addition of a bonus wiki-assignment that had students expand existing testing related pages.

The pass criteria and point distribution between different activities (Table 3) also remained roughly the same with some additional points becoming available from the new wiki-assignment. Weekly assignments were previously an optional part of the course as the students could accumulate enough points to pass the course without actively taking part in doing the weekly assignments. In the updated version of the course weekly assignments were changed to being compulsory part of the course, requiring a total of 15 points to be able to pass the course. Likewise, students would have to obtain half of the total points available from both the course project and the exam to pass the course.

Table 3. Course grading categories and available points during year 2023.

Category	**Available points**	**Compulsory**
Exam	30	Yes
Project	30	Yes
Weekly assignments	30	Yes
Guest lecture	10	No
Wiki-assignment	5	No

5.2 Environment and tools

This section details the environment and tools used to implement the course *Fundamentals of Software Testing*. The goal is to briefly introduce the core tools that make up the course environment and describe some of their capabilities. As an automated assessment tool, CodeGrade itself includes many features to benefit both students and course instructors, making it the main point of discussion. All the course programming assignments are implemented with CodeGrade capabilities in mind and the students are not required to set up a special programming environment. However, for some of the weekly assignments and for the course project it is still recommended to set up a local testing environment.

5.2.1 Moodle

Moodle is an extendable open source learning management system (LMS) used at LUT University courses that is intended to provide an interactive online workspace that consolidates all course material, assignments and even exams to one location. Several courses conducted by the software engineering department accept external students and Moodle offers an easy way to authenticate users and enrol on a course even as a student that is not using dedicated enrolment systems at LUT.

Moodle allows the usage of *Learning Tools Interoperability* (LTI) protocol to add external add-on activities, which CodeGrade makes use of to integrate with LMSs such as Moodle, Blackboard, Brightspace, Canvas and Sakai (CodeGrade, 2024a; Moodle, 2024). This way Moodle accounts are automatically synchronized with CodeGrade through a shared session and there is no need to separately log in to CodeGrade whenever working on programming assignments. This also allows bi-directional communication between the two services to synchronize student grades automatically. Other shared data between LMS and CodeGrade include username, full name, email address, course name, assignment name and -state (CodeGrade, 2024).

Moodle also provides other functionality that makes course management easier and facilitates better learning. It’s possible to hold online discussions in the form of forums and chats where the students can seek help from other students or course personnel. Moodle offers some built-in features for collecting and viewing data, as well as creating and exporting reports to different formats. These features make it easy to track student progression during the course and in the context of this thesis, they also make it possible to analyze different aspects of the course such as weekly assignments and course activity to then draw conclusions on the implementation.

5.2.2 VS Code

Popularity of IDEs heavily correlates with popularity of different programming languages but if not considered, Visual Studio and the more lightweight Visual Studio Code (VS Code) hold the number one spot among most popular IDEs. Amongst programmers that have chosen Java as their main programming language, the share of IDEs looks very different with IntelliJ holding top spot with 78% share and VS Code with 6% share of Java developers. (Carbonnelle, 2023)

NetBeans used to be the recommended IDE to be used during the course but was changed and demo videos updated to support VS Code instead. For this course the IDE choice is not a critical factor as the role of it is to support simple testing activities locally on the students' own system as opposed to larger projects. Because of its overall popularity it is also very likely that students would have previous experience with VS Code or would end up using it in the future. Like other popular IDEs, VS Code includes many features that make coding easier especially for beginners. Syntax highlighting, code completion and -hints, debugging capabilities, extensibility and its ease of use make it suitable for this use case. The current requirements for the IDE are simple as it should at a minimum: (1) support testing with JUnit, and (2) allow students to run Java code locally.

5.2.3 CodeGrade

CodeGrade is an automated assessment tool and a blended learning application tailored for use in programming education. It originated as a course project at the University of Amsterdam, motivated by firsthand experiences both as a student and as a teaching assistant on computer science courses. Traditional grading methods were observed to be insufficient for providing meaningful feedback and from the TA perspective the grading process was seen as being too rigid, causing unnecessary overhead. CodeGrade includes variety of different features that are used to make programming education easier for both students and instructors. These are features such as grading related tools allowing creation of automatic or manual grading rubrics or checking the authenticity of submissions via plagiarism detection tools. Instructors are also able to benefit from CodeGrade analytics dashboard for optimization purposes. This section will focus on the core features used in the assessment of programming assignments and some of the more important features the students are in contact with. These are features such as *Code viewer* and *AutoTest*. (CodeGrade, 2024b)

CodeGrade is built on the following three core technologies: (1) Vue.js, (2) Flask, and (3) PostgreSQL. Vue is a front-end JavaScript framework used to build the user interface of CodeGrade. Flask, which is a Web Server Gateway Interface (WSGI) framework, is used to create the back-end of CodeGrade and finally, PostgreSQL is used as the database. (CodeGrade, 2024c)

Code viewer (Figure 3) is an online grading environment that includes functionality for both teachers and students. For the students it is the main view they are presented to with their submissions. Code viewer consists of three main sections that each provide different functionality related to the submission: (1) Code, (2) Feedback Overview, and (3) AutoTest. *Code*-section displays the submitted file with proper language specific highlighting and supports inline feedback. It also includes a hierarchical file selection used to navigate between submitted files and displays files that can be automatically generated during AutoTest execution. *Feedback*-section simply displays all feedback given on an assignment including inline- and general feedback. *AutoTest*-section is used to display the result of automated assessment that the submission is tested against.

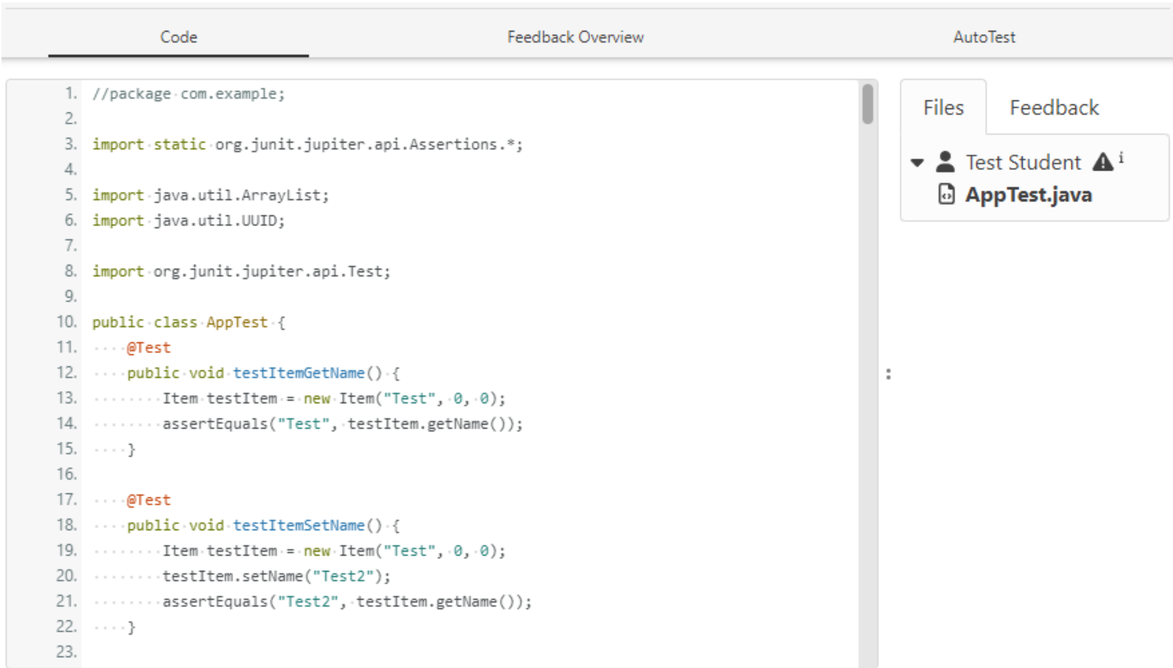

Figure 3. Code viewer user interface.

AutoTest (Figure 4) is the system responsible for automated assessment and grading of students' programming assignments. With AutoTest, each student submission uses their own individual Virtual Machine (VM) that runs on the latest Long Term Support (LTS) version of Ubuntu. The VM environment itself has a wide variety of pre-installed software commonly used to grade student submissions. Additionally, the environment comes with wrapper scripts that support several different testing frameworks for different languages. For

Java, both JUnit 4 and -5 are included but need to be installed separately with the help of a wrapper script. AutoTest starts with a setup phase and creates a snapshot that is then used to initialize separate containers for each testing phase or category. Setup phase is intended to initialize the environment so that all the required files and versions of grading tools are in order before running tests. Tests are built as combination of categories and related grading rubrics. Each category then runs a set of tasks or smaller sub-tests determined by the visual coding blocks and finally outputs a score. (CodeGrade, 2024d)

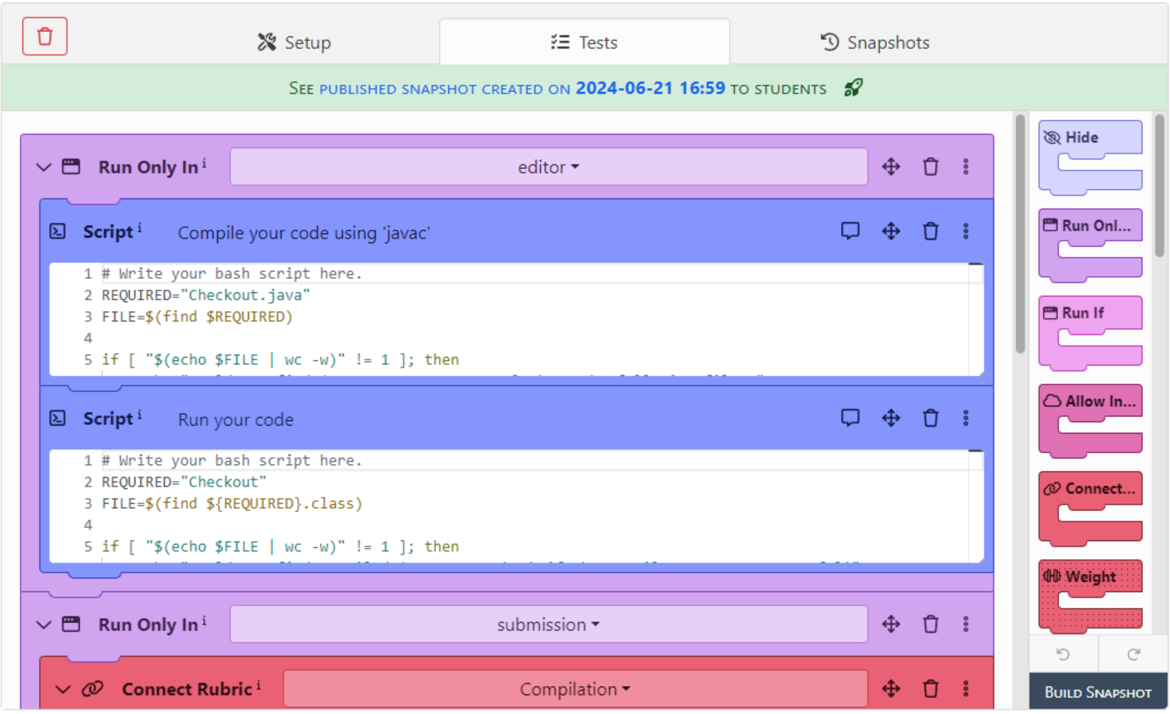

Figure 4. Example test structure in AutoTest V2. Assignment from CodeGrade community library.

5.3 Assignments

In their research Kollanus et al. (2008) describe their successful experience in incorporating TDD on a software testing course in the University of Jyväskylä and conclude that incorporating TDD concepts to the curricula should happen early on but not too early; meaning that TDD combined with challenging programming tasks can have an unintended effect of overwhelming the students. Following their conclusion, to fit an introductory software testing course, the course assignments were built to introduce testing concepts but at a level where the concepts are easy to understand and don't require the students to produce complicated code together with their tests. This was further reasoned with the limited scope of the course and the number of other additional weekly tasks.

First couple of weeks' lecture content consisted mostly of introduction of software testing as a subject, different testing phases and terms related to testing. Based on the lecture topics it was seen appropriate to introduce the students to how unit testing works in practice and so the first programming assignments were more of an introduction to using JUnit and setting up the local test environment. In TDD fashion, in the first programming assignment the students were asked to implement a Body Mass Index (BMI) calculator based on already provided set of unit tests. The assignment included a ZIP-file that contained two different test classes corresponding to different grades obtainable from completing the assignment. The student could choose to only complete the BMI-calculator as defined by the first set of tests and obtain one point from the assignment or they could complete the complete calculator as defined by the second set of tests to obtain full points from the task. The assumption was that students already had previous experience from Java programming but not on unit testing or JUnit. This approach would give them a chance to see the basic structure of simple JUnit tests and allow them to ensure they are able run JUnit in their local environment. During the second week the concept was flipped around, and the students instead had to make their own unit tests utilizing JUnit. The assignment description and assessment process were set up in a way that ensured the students use proper unit-testing practices and build their unit tests using proper syntax.

In the assignment 3-2 the students were asked to create unit tests for a larger example program consisting of several different classes. The assignment consisted of two separate parts that were both part of the grading process: (1) creating unit tests that are required to pass and, (2) creating unit tests that should fail, catching errors in the program code. The first part required running students' unit test against an example solution to ascertain correctness of their unit tests. In the second part the tests were ran again but this time against a solution where defects were introduced manually. The goal was then to find these defects by not being able to visually detect them due to not having access to the version of the program with bugs introduced. Assignment 6-2 had students also hunting for bugs but this time the students had access to the faulty program code and were able to find bugs using program description, comments and CodeGrade hints.

Assignment 4-2 introduced a practical example of code coverage. In this assignment the students were to conduct testing on a program with the goal of achieving near perfect coverage percentage. The assignment itself was structured in a way that allowed the student to reach almost perfect coverage by utilizing CodeGrade and JaCoCo to visually display achieved coverage percentages for each class and method. Finally, assignment 5-2 had students creating test stubs that allowed existing unit tests to pass.

5.4 Assessment of assignments

Assessment process of the programming assignments was implemented using three primary technologies: (1) Semgrep, (2) JUnit, and (3) JaCoCo. Technologies used to assess different course programming assignments and their inclusion in weekly assignments can be seen in Table 4. Static assessment of program code and tests was accomplished using *Semgrep*, a relatively new static code analysis tool first released in 2020 that supports a wide variety of different programming languages (Semgrep, 2024). *JUnit* is an open-source testing framework for Java programming language that is able to integrate with different build tools such as Maven, Gradle or Ant (JUnit, 2024). *JaCoCo* is a code coverage tool for Java that is paired together with JUnit to measure test adequacy.

Table 4. Technologies used with each CodeGrade programming assignment.

Assignment	Semgrep	JUnit	JaCoCo
1-2	✓	✓	
2-2	✓	✓	
3-2		✓	
4-2		✓	✓
5-2		✓	
6-2		✓	

Semgrep is used as a static code analysis tool on first two weeks of programming assignments to ensure the students learn proper unit test structure and related conventions. After first two programming assignments the importance of enforcing a certain test structure was not seen as being the primary consideration, but the correctness of submissions was still ensured with more detailed instructions and by formulating assignments in a way that students had no choice but to come up with the right solutions to achieve a good grade. Semgrep is very effective when the assignments follow a similar structure but struggles with assignments that have multiple possible solutions (Avramov, 2022).

JUnit is the primary technology used in the assessment process and is used in the AutoTest implementation of all course assignments. It is also the main tool used by students during the course to construct their own tests. JUnit version 5 (JUnit 5) is used to execute a set of test cases either provided by the student or uploaded to the assessment environment by the instructor. To evaluate student submissions, a custom wrapper script is used to run the command line version of JUnit with the defined test suite to generate an XML-report of the test results. The report is then parsed to a CodeGrade suitable format and output to be used by the AutoTest “Custom Test”-block (Figure 5). In the case of failure each individual test result also provides information on the error and may also include other instructor added hints. The students were typically awarded points for each passing unit test.

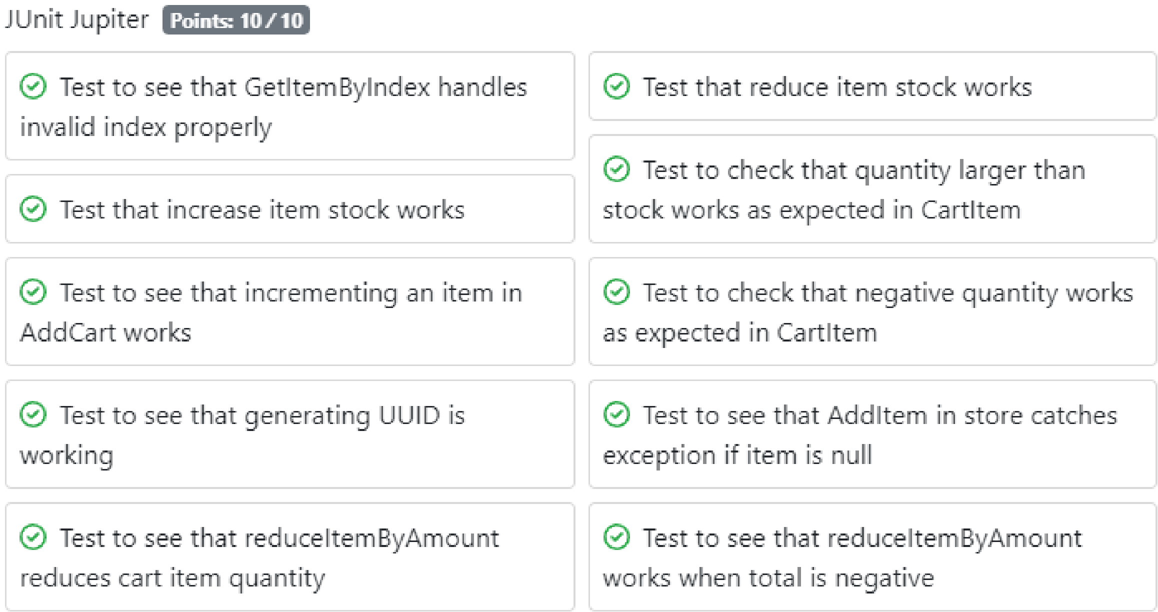

Figure 5. Example AutoTest result of student submitted tests. Custom Test -block displaying JUnit test results.

JaCoCo is used to evaluate the adequacy of the students' test suite by calculating a coverage metric that indicates which parts of the code were covered by the tests. JaCoCo can be used as a direct indicator on test adequacy but during the course it was used to teach students on how code coverage works together with their JUnit tests. It works via class file instrumentation using an agent together with the JUnit tests. Running JUnit test suite together with JaCoCo generates a coverage report that can be visually examined to locate different areas of improvement. This was previously possible in CodeGrade as the AutoTest V1 allowed the students to view the generated visual representation of the coverage report if the report was moved to a special AT_OUTPUT-directory that is accessible to students (CodeGrade, 2021). However, with AutoTest V2 this was no longer possible and customizing a wrapper script was necessary to generate a visually pleasing representation of the coverage achieved. The wrapper script works in a similar fashion to the one used with JUnit, the difference being in having to inject the JaCoCo agent and parsing the generated coverage report to a representable format (Figure 6).

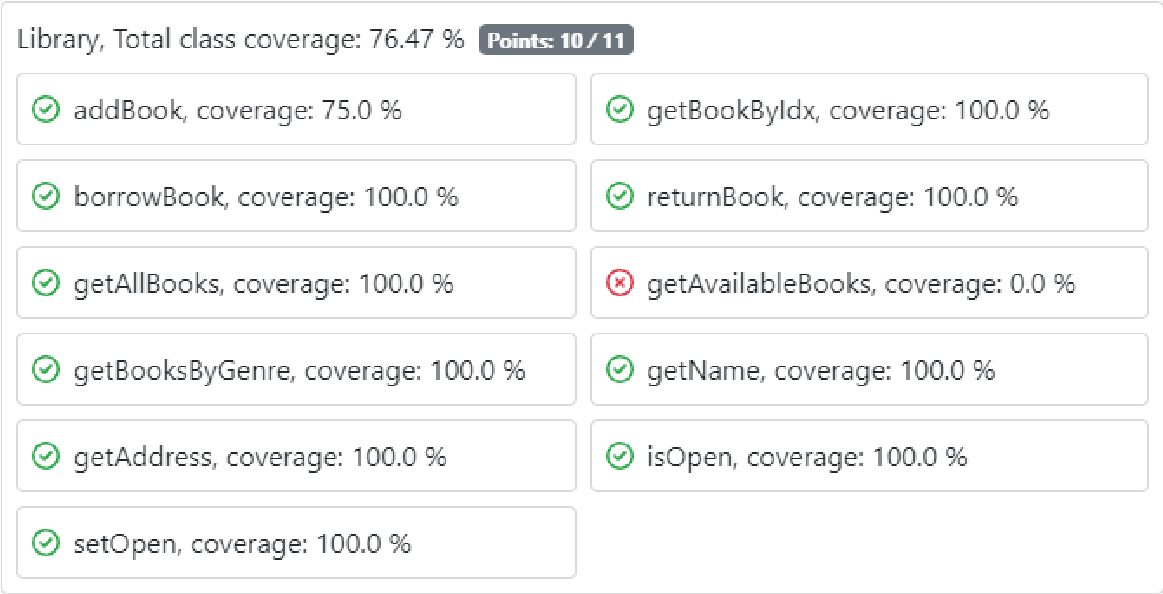

Figure 6. CodeGrade submission code coverage example using JaCoCo and JUnit.

6 Results

The purpose of this chapter is to objectively present results of the research based on the feedback obtained from course participants in the form of a survey and based on data collected from course Moodle page during the 2023 fall iteration of the course. The intention of these results is to communicate how the change in course assignment structure was perceived among the students and to compare trends in programming assignment submissions between the two versions of the same course, i.e. course before and after introduction of CodeGrade.

6.1 Survey

Data was collected in a form of a survey that was open to all participants after the course had officially concluded and rewarded a point towards their final grade. A total of 79 answers were collected out of 168 total participants which equates to roughly 47% of students that submitted an answer to the survey. The survey itself included not just feedback on the CodeGrade assignments but rather on the course as a whole – that is, polling general feelings on the course topics, motivation and collecting free-form input on what went well or what aspects need more improvement. This section limits presenting the survey results to areas that directly relate to programming assignments.

The questions included both open-ended questions and questions where students were asked to reply on a Likert-scale with a value ranging between 1-5. Open ended questions are summarized with some of the more interesting quotes brought up as an example.

Q: *The workload relative to the study credits awarded was.* (Type: Likert. 1 very light; 3 appropriate; 5 very heavy)

The question itself does refer to the workload of the complete course, not just programming assignments, but can still be used as a good indicator on whether the course assignment difficulty was seen as adequate. Average of the answers landed almost exactly in the middle with value of 3.1 and standard deviation of 0.6 (Figure 7). From the students' point of view the total course workload was seen as appropriate.

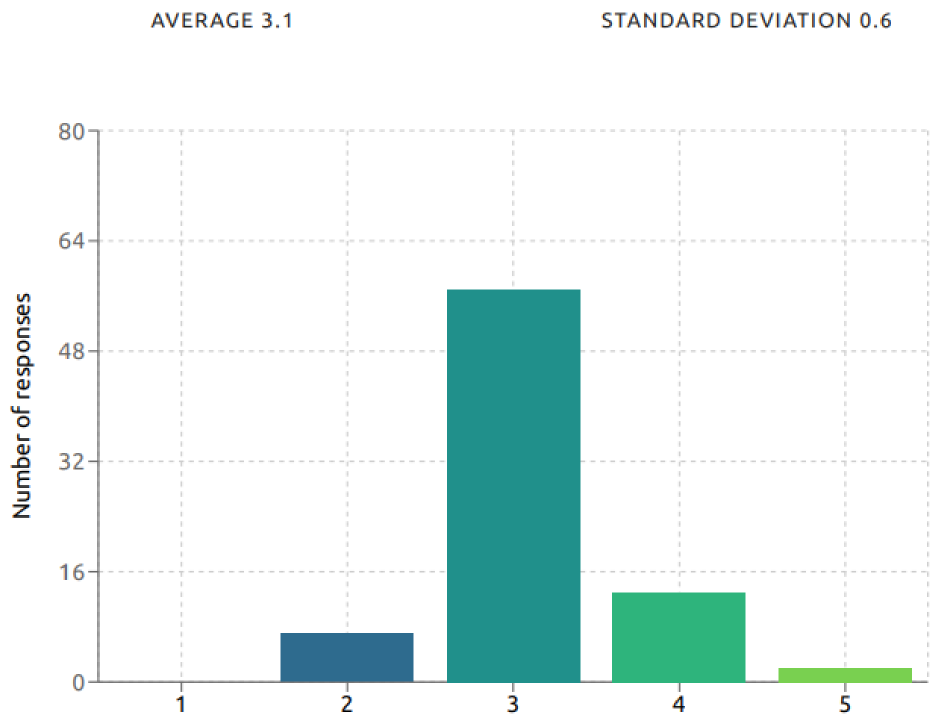

Figure 7. Course workload as estimated by students.

Q: *What aspects of the course need most improvement?* (Type: Open)

Regarding weekly programming assignments one of the more requested improvements was that the assignments should have more study materials and practical examples available, especially during the first few weeks when JUnit is being introduced for the first time. The experience with CodeGrade itself seems to have been a positive one but some respondents wished for more assignment variety in terms of used technologies. One answer stated:

"Programming assignment instructions especially on the first weeks could have been clearer."

Some students seemed to also experience problems setting up their local testing environment and hoped for additional material or step by step guides on how to make JUnit work on their own machine. Especially for macOS users setting up the testing environment seemed to create some unique challenges.

Q: *What was best about this course?* (Type: Open)

Programming assignments and CodeGrade received a lot of praise and many respondents felt that the programming assignments ended up being their favorite part of the course. Course content and assignments were seen as easy to understand and some answers mentioned liking the weekly assignment structure because they could still get something out of the course and pass without being confident in their programming abilities. A respondent mentioned:

"The weekly exercises were fun to do and the CodeGrade system was implemented well. The project goal was also very clear. The teaching videos were informative too."

Q: *How useful would you consider the following parts of the course?* (Type: Likert. 1 not useful; 5 very useful)

Several aspects of the course were graded (Figure 8) with the topics of interest for this thesis being exercise tasks and the CodeGrade submission system. Exercise tasks were graded with an average of 4.3 and CodeGrade received an average grade of 4.2. The least favorite part of the course was overwhelmingly named to be the optional wiki-assignment which was added as a new task for this iteration.

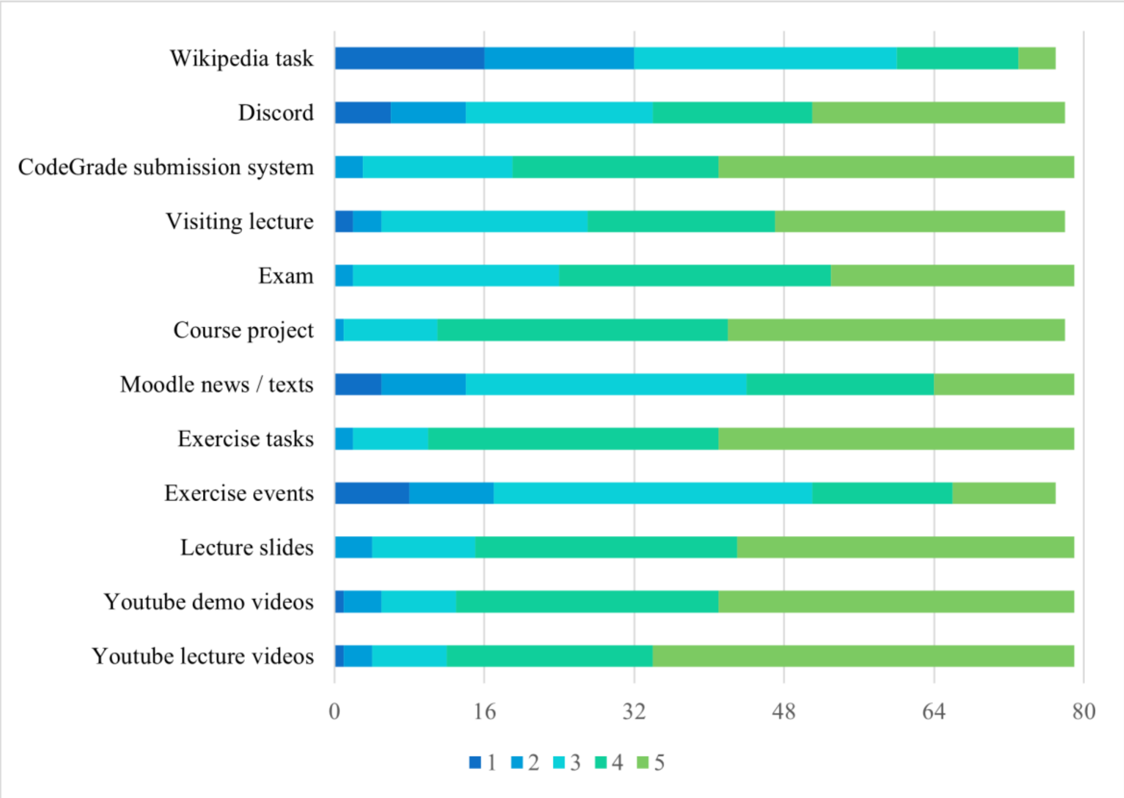

Figure 8. Course topic usefulness according to respondents.

6.2 Course assignments

Course assignment data was extracted from the course Moodle page and analyzed to gain insight into what changes the programming assignments have gone through in terms of attained grades and submission rates for each assignment. Data was obtained from both before implementing the new programming assignments and after updating programming assignments to support CodeGrade as an automated assessment tool. This section will present and compare the data obtained from these two course iterations.

As was stated in section *5.1 Updated course structure*, the new course assignments were changed into a more consistent format. Instead of having a constant mix of writing and programming assignments, the weekly assignments had one of each type, i.e. quiz, programming assignment and a writing task each week. The average submission ratio between writing assignments remained roughly the same (0.7 in 2022 compared to 0.67 in 2023) (Figure 9). However, the average submission ratio of programming assignments increased noticeably from 0.41 to 0.59. As the previous course implementation did not include quiz-type assignments, no comparison can be made for quiz-activities.

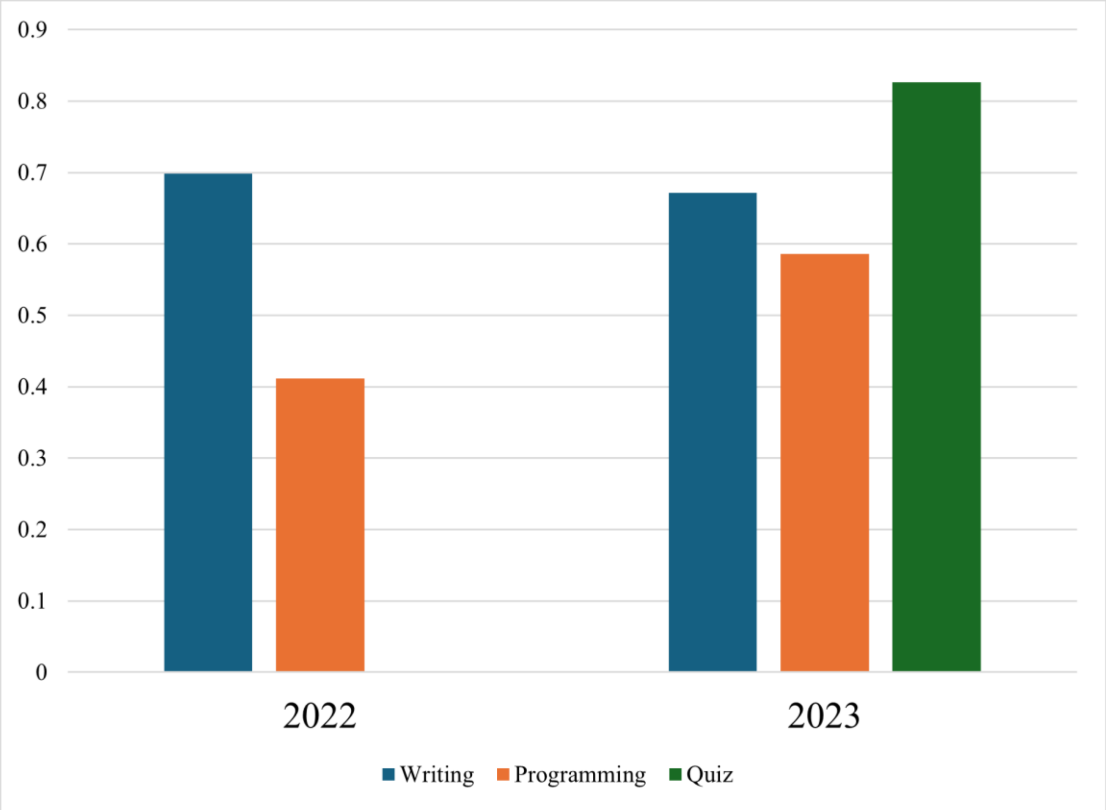

Figure 9. Submission ratio of different assignment types during old (2022) and updated (2023) versions of the course.

Submissions rates of all programming assignments during both the old and new version of the course can be seen in *Figure 10*. A downwards trend in the submission ratio can be seen in the submission data of both course implementations. However, the 2022 version of the course had a more dramatic decline as compared to the 2023 version while also experiencing larger deviation between submission ratios of different assignments. The old course had its peak submission ratio of 0.7 during the submission of first programming assignment and the lowest submission ratio of 0.18 during the fifth week of the course. On the other hand, the updated course experienced its highest peak of 0.67 also during the first programming assignment of the course but followed a very linearly decreasing path with the lowest point being the last programming assignment with a value of 0.49.

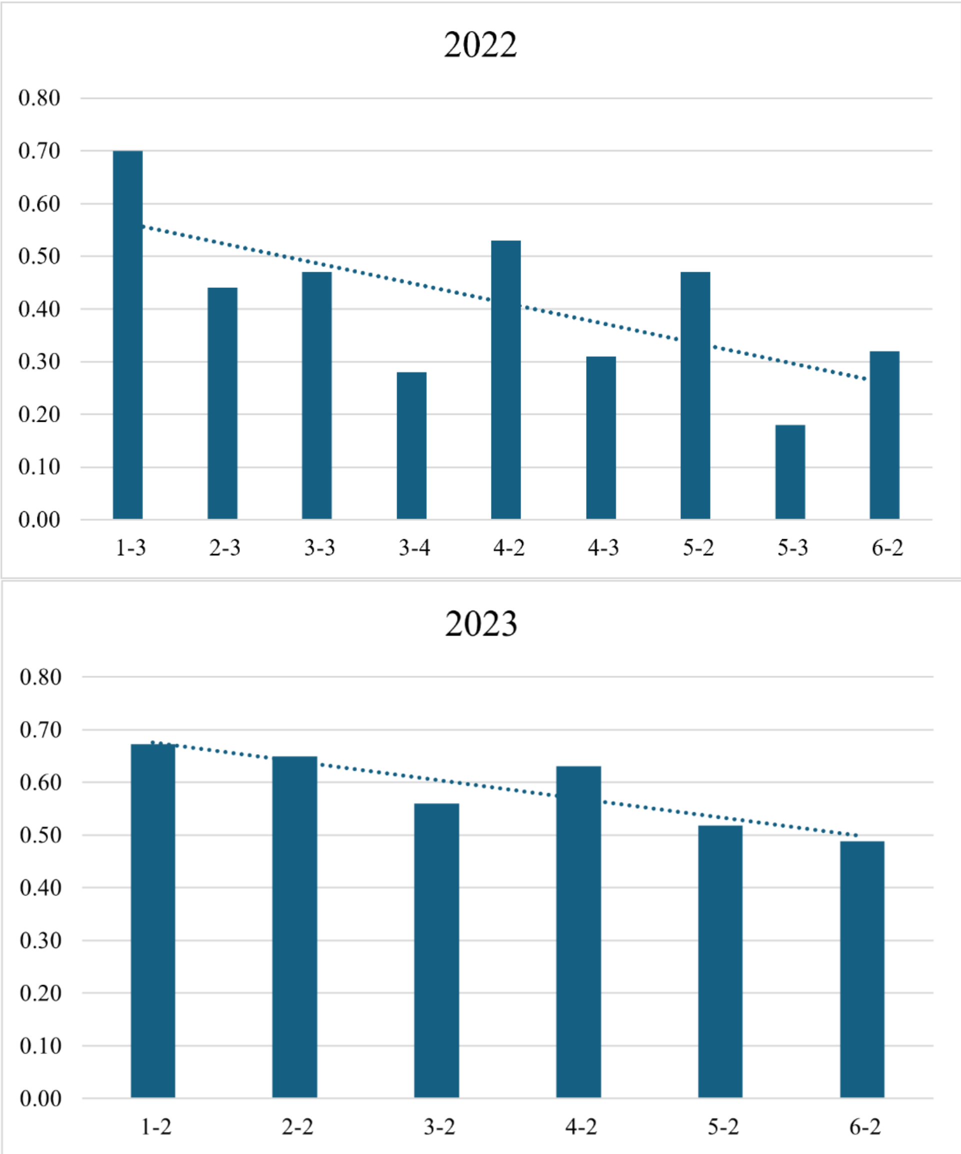

Figure 10. Submission rate of all course programming assignments during old (2022) and updated (2023) versions of the course.

General data for all the course assignments can be seen in Figure 11. Average number of submissions is only calculated for CodeGrade assignments that support unlimited submissions, whereas quizzes and essay-type assignments are limited to only one submission for each student. Assignment 3-2 stands out as having especially high number of average submissions per student. The submission ratio and average score for this assignment also indicates slight increase in difficulty. Average score for all weekly assignment submissions reached nearly 90% of the maximum points available which indicates that the difficulty of the assignments was certainly not too difficult for those who made an attempt at the assignment.

Assignment	Submission ratio	Average score	Max points	% of max	Avg. submissions
Quiz #1	0.91	0.87	1	86.67 %	
1-2	0.67	1.74	2	87.12 %	3.49
1-3	0.77	1.92	2	96.12 %	
Quiz #2	0.84	0.88	1	87.73 %	
2-2	0.65	1.95	2	97.29 %	2.60
2-3	0.70	1.76	2	87.82 %	
Quiz #3	0.83	0.91	1	90.79 %	
3-2	0.56	1.45	2	72.27 %	7.72
3-3	0.76	1.78	2	88.78 %	
Quiz #4	0.83	0.92	1	92.09 %	
4-2	0.63	1.61	2	80.66 %	4.42
4-3	0.70	1.68	2	83.90 %	
Quiz #5	0.80	0.95	1	95.37 %	
5-2	0.52	1.88	2	93.88 %	3.49
5-3	0.54	1.63	2	81.67 %	
Quiz #6	0.75	0.92	1	91.90 %	
6-2	0.49	1.97	2	98.29 %	2.85
6-3	0.57	1.93	2	96.32 %	

Figure 11. Assignment data for all course assignments during year 2023. Submission ratio, average score of each assignment, maximum points available, percentage of maximum points attained and average number of submissions for all programming assignments included in the figure.

6.3 Course demographics

Course demographics for both versions of the course are presented as contextual data in order validate the course assignment data used in previous sections. A Largely different course demographic could influence the collected assignment data if, for example, the number of students from other faculties with less programming experience suddenly increased.

According to course data collected from the Moodle pages of each yearly iteration of the course Fundamentals of Software Testing, the faculty split between students remained roughly the same while the total student numbers increased from 142 to 168 students (Figure 12). The clear majority of the students were part of the LUT School of Engineering Sciences, followed by FiTech, School of Energy Systems, and finally School of Business and Management, listed in decreasing order based on number of students. School of Engineering Sciences includes variety of different areas of expertise, software engineering being one of them (LUT University, 2024). FiTech on the other hand is an external network university collaborative with several major Finnish universities participating and offering a selection of technology related courses that range from introductory courses all the way to more advanced courses (FiTech, 2024). Based on the course student demographics it is relatively safe to conclude that the faculty split on each year should have no significant impact on assignment data.

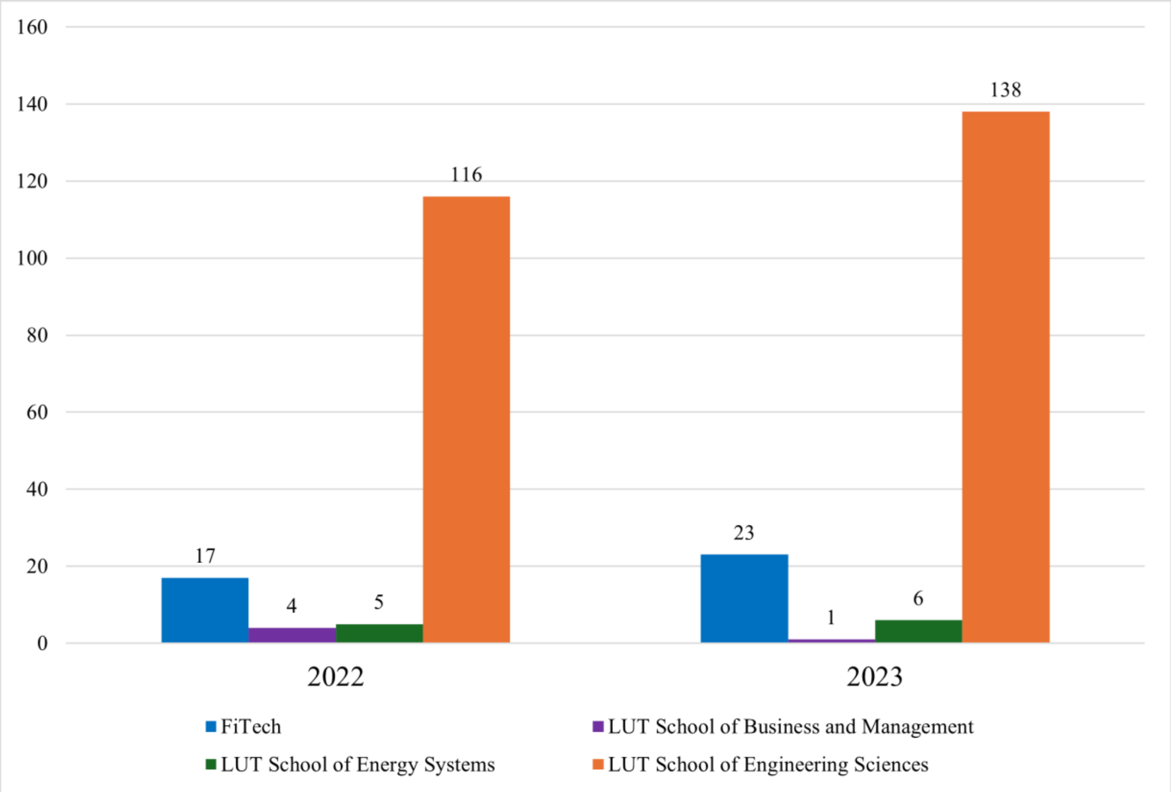

Figure 12. Student numbers of each faculty during years 2022 and 2023.

7 Discussion

The survey results indicate that the updated programming assignments and the introduction of CodeGrade were received positively. Submission ratio of programming assignments saw drastic improvements over the previous implementation and the collected responses as well as course data indicate that the students were motivated to attempt the programming assignments and receive a good grade. One of the initial goals when creating new programming assignments was to engage the students with a low barrier to entry so that they no longer opt to skip the assignment completely in favour of other non-programming assignments. Previously, with the weekly assignments being a non-compulsory part of the course, the students were able to avoid the more difficult looking assignments altogether. However, with the restructuring of assignments and the introduction of automated assessment, the students seemed more encouraged to make an honest effort at the programming assignment. To achieve this, the students were permitted an unlimited number of submissions which encouraged them to turn in a solution even if they did not perceive it as being complete. This way the students were able receive a small number of points even if they did not complete the entirety of the assignment and were rewarded for their efforts accordingly. It is good to note that the drastic increase in assignment submission ratio, i.e. students that had at least one submission on an assignment, is partly explained due to lowered difficulty in receiving partial grades which in turn worked to encourage the students to keep improving their submission. This way the students were provided with immediate positive feedback that their new solution did in fact perform better in comparison to their previously submitted version.

Many survey respondents expressed their satisfaction in the programming assignments and how they supported learning through more concrete examples. On the contrary, some explicitly stated that they wished for more practical examples that supported programming assignments. The introduction of JUnit at the start of the course was in some cases seen as being too abrupt as the course material and lecture content had not yet been updated to reflect new programming assignments. While the students were provided with hints and external resources to help them solve the current problem, some still wished for more direct examples, especially on the usage of different JUnit assertions.

Selection of technologies for automated assessment was initially motivated by related research on different static- and dynamic assessment tools, particularly those related to prior experiences using CodeGrade in an educational setting. One such case is Avramov's (2022) experience using a combination of CodeGrade, JUnit and Semgrep to grade programming exams. Selection of tools was further limited by current CodeGrade capabilities as well as recommended technologies showcased in CodeGrade documentation and blog posts (CodeGrade, 2024a; CodeGrade, 2021). Semgrep was first used to ensure that the students followed recommended practices when building their test cases, but this was quickly deemed to cause unnecessary friction if the used patterns were too rigid. Programming exams are generally graded while considering coding style and other structural specifications. However, in the case of assessing unit tests, the role of a static assessment tools was regarded more as a supporting tool and the students were allowed to design their test cases more freely. Even in the absence of static code analysis tools, the quality of submissions remained consistent.

VS Code was included as the recommended way for students to develop and test their test cases in a local environment. It was still possible for the students to complete the assignment by only utilizing the CodeGrade built-in code editor, but setting up a local environment was recommended in order to teach the students how they can make use of standard development tools such as debugger to find flaws and improve their code. However, some students reported difficulties in setting up their local testing environment, especially macOS users struggled finding supporting material to help them get started. As a solution to this problem, the course material could be further improved to provide means for students to self-diagnose and solve the issues they are having with their local development environment. After the initial hurdles, the experience using VS Code for programming assignments was regarded positively.

It is difficult to accurately estimate the success of CodeGrade implementation and its capability in assessing student submissions because the introduction of CodeGrade was not the only new change to the course. The course itself went through small structural changes and even in the case of programming assignments the contents and grading criteria were changed considerably. This makes it difficult to directly compare previous grades and draw conclusions from them. Furthermore, the AA environment created some limitations in terms of assignment design as the assessment of more creative designs was often very difficult or created opportunities for dishonesty in terms of cheating the system. For example, during assignment 4-2, which presented a practical example of code coverage, it became apparent that CodeGrade would not be able to display the automatically generated visual report on achieved coverage and so a workaround was used to generate a coverage report using CodeGrade functionality. According to the survey results and feedback received during course exercise events, the experience using CodeGrade was very smooth and caused little to no friction. The biggest cause of confusion was the lack of useful feedback in cases where the submitted program did not compile due to presence of errors in the student's code.

In the future, the assignments should be improved to include more tools to provide feedback on errors and coding style. Currently, the students are able to use the general error messages to pinpoint problems, but in some cases, this is a major cause for confusion. There is an opportunity to improve the current assignments by including style enforcement with static analysis tools. One weakness of JUnit is its inability to compile in the presence of syntactic errors (Avramov, 2022). With static analysis tools it's possible to provide meaningful feedback in such cases where compilation is not possible. Another area of improvement is the course material and its linking to programming assignments through practical examples. This is an area that is likely to improve in the future versions of the course. Additionally, the assignments could be expanded to include different kinds of testing methods and tools. For example, the use of mutation testing in assessment of programming assignments holds potential for further research.

8 Conclusion

The objective of this thesis was the implementation of a newly adopted automated assessment tool, CodeGrade, on an introductory level software testing course. In the process, related literature was reviewed to gain insight into different methods of assessing programming assignments, their benefits and possible drawbacks. Course programming assignments were updated and migrated to use CodeGrade, after which a survey was conducted to find out students' attitudes and experiences using CodeGrade during the course.

Increasing student numbers on software engineering courses resulted in the need to introduce automated assessment to answer issues with grading consistency, availability and speed in receiving quality feedback. CodeGrade is an automated assessment system that, while being a new addition to LUT University ran software engineering courses, had been successfully piloted on some of the introductory level courses but had not yet been applied in the context of software testing. Software testing is an important part of software engineering education but also creates some unique challenges in terms of how to effectively evaluate the students' testing skills.

A survey was conducted that asked the respondents to evaluate different course aspects on a Likert-scale with values ranging between 1–5. The usefulness of CodeGrade received an evaluation of 4.2 and together with free-form feedback it is clear that the experience using CodeGrade as a programming assignment assessment system was received well. The total workload of the course was evaluated with an average score of 3.1, which indicates that the assignment difficulty was seen as adequate from the students' point of view. Additionally, the submission ratio of assignments improved greatly compared to the previous implementation of the course and remained consistent from start to finish.

Large portion of feedback was associated with the need for additional supporting learning material, especially during the first couple of weeks of the course. Future improvements should focus on providing more practical examples and linking the course materials more tightly to the assignments. It is also important to consider how to introduce different aspects of software testing as the programming assignments still largely focus on unit testing topics. While the workload of the course was evaluated to be appropriate there is also some room to consider adding more in-depth assignments.

References

Aaltonen, K., Ihantola, P., Seppälä, O., 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. https://doi.org/10.1145/1869542.1869567

Ala-Mutka, K.M., 2005. A Survey of Automated Assessment Approaches for Programming Assignments. Computer Science Education vol. 15, pp. 83–102. https://doi.org/10.1080/08993400500150747

Arifi, S.M., Abdellah, I.N., Zahi, A., Benabbou, R., 2015. Automatic program assessment using static and dynamic analysis, Third World Conference on Complex Systems (WCCS). Presented at the 2015 Third World Conference on Complex Systems (WCCS), pp. 1–6. https://doi.org/10.1109/ICoCS.2015.7483289

Avramov, V.R., 2022. Automated grading of programming exams with JUnit and Semgrep.

Beller, M., Bholanath, R., McIntosh, S., Zaidman, A., 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software, IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 470–481. https://doi.org/10.1109/SANER.2016.105

Berry, R.E., Meekings, B.A.E., 1985. A style analysis of C programs. Commun. ACM 28, 80–88. https://doi.org/10.1145/2465.2469

Brocke, J. vom, Hevner, A., Maedche, A., 2020. Introduction to Design Science Research. pp. 1–13. https://doi.org/10.1007/978-3-030-46781-4_1

Carbonnelle, P., 2023. TOP IDE Top Integrated Development Environment index [WWW Document]. URL https://pypl.github.io/IDE.html (accessed 6.19.24).

Cheang, B., Kurnia, A., Lim, A., Oon, W.-C., 2003. On automated grading of programming assignments in an academic institution. Computers & Education 41, 121–131. https://doi.org/10.1016/S0360-1315(03)00030-7

Checkstyle, 2024. Standard Checks [WWW Document]. URL https://checkstyle.sourceforge.io/checks.html (accessed 1.29.24).

Chen, P.M., 2004. An automated feedback system for computer organization projects. IEEE Transactions on Education 47, 232–240. https://doi.org/10.1109/TE.2004.825220

CodeGrade, 2024a. The CodeGrade Documentation [WWW Document]. URL https://docs.codegra.de/ (accessed 6.19.24).

CodeGrade, 2024b. History of CodeGrade [WWW Document]. URL https://docs.codegra.de/about/history.html (accessed 6.23.24).

CodeGrade, 2024c. Used Technologies [WWW Document]. URL https://docs.codegra.de/about/technologies.html (accessed 6.23.24).

CodeGrade, 2024d. AutoTest [WWW Document]. URL https://docs.codegra.de/user/autotest/overview.html (accessed 6.23.24).

CodeGrade, 2021. Testing the Tests: Autograding student unit tests in Java assignments (using JUnit code coverage) [WWW Document]. URL https://www.codegrade.com/blog/testing-the-tests-autograding-student-unit-tests-in-java-assignments-using-junit-code-coverage (accessed 6.23.24).

de Souza, D.M., Oliveira, B.H., Maldonado, J.C., Souza, S.R.S., Barbosa, E.F., 2014. Towards the use of an automatic assessment system in the teaching of software testing, IEEE Frontiers in Education Conference (FIE) Proceedings, pp. 1–8. https://doi.org/10.1109/FIE.2014.7044375

Edwards, S., 2014. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance.

Edwards, S., 2003a. Improving student performance by evaluating how well students test their own programs. J. Educ. Resour. Comput. 3, https://doi.org/10.1145/1029994.1029995

Edwards, S., 2003b. Rethinking Computer Science Education from a Test-first Perspective.

FITech, 2024. Opiskelu FITechissä [WWW Document]. URL https://fitech.io/fi/opiskelu-fitechissa/ (accessed 6.15.24).

Forišek, M., 2006. Security of Programming Contest Systems. Information Techonologies at School.

Galan, D., Heradio, R., Vargas, H., Abad, I., Cerrada, J.A., 2019. Automated Assessment of Computer Programming Practices: The 8-Years UNED Experience. IEEE Access 7, 130113–130119. https://doi.org/10.1109/ACCESS.2019.2938391

Hemmati, H., 2015. How Effective Are Code Coverage Criteria? IEEE International Conference on Software Quality, Reliability and Security, pp. 151–156. https://doi.org/10.1109/QRS.2015.30

Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. Design Science in Information Systems Research. MIS Quarterly.

Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O., 2010. Review of recent systems for automatic assessment of programming assignments, Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling'10. https://doi.org/10.1145/1930464.1930480

Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness, Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. Association for Computing Machinery, New York, NY, USA, pp. 435–445. https://doi.org/10.1145/2568225.2568271

Ivankovic, M., Petrovic, G., 2019. Code Coverage at Google. https://doi.org/10.1145/3338906.3340459

Jackson, D., Usher, M., 1997. Grading student programs using ASSYST, Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education - SIGCSE '97, pp. 335–339. https://doi.org/10.1145/268084.268210

PMD Source Code Analyzer Project, 2024. Java Rules [WWW Document]. URL https://docs.pmd-code.org/latest/pmd_rules_java.html (accessed 1.29.24).

JUnit, 2024. JUnit 5 User Guide [WWW Document]. URL https://junit.org/junit5/docs/current/user-guide/ (accessed 6.26.24).

Kay, D.G., Scott, T., Isaacson, P., Reek, K.A., 1994. Automated grading assistance for student programs. SIGCSE Bull. 26, 381–382. https://doi.org/10.1145/191033.191184

Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., Le Traon, Y., 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. Empir Software Eng 23, 2426–2463. https://doi.org/10.1007/s10664-017-9582-5

Kyrilov, A., Noelle, D.C., 2015. Binary instant feedback on programming exercises can reduce student engagement and promote cheating, Proceedings of the 15th Koli Calling Conference on Computing Education Research, pp. 122–126. https://doi.org/10.1145/2828959.2828968

Lenarduzzi, V., Pecorelli, F., Saarimaki, N., Lujan, S., Palomba, F., 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. Journal of Systems and Software 198. https://doi.org/10.1016/j.jss.2022.111575

Moodle, 2024. LTI External tools - MoodleDocs [WWW Document]. URL https://docs.moodle.org/404/en/LTI_External_tools (accessed 6.19.24).

LUT University, 2024. Faculties [WWW Document]. URL https://www.lut.fi/en/about-lut/faculties (accessed 6.16.24).

Mäkelä, P., 2021. Selection process of an automated assessment system at LUT University. Automaattisen tarkastustyökalun valintaprosessi LUT-yliopistossa.

Malmi, L., Korhonen, A., Saikkonen, R., 2002. Experiences in automatic assessment on mass courses and issues for designing virtual courses, Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, pp. 55–59. https://doi.org/10.1145/544414.544433

Maximilien, E.M., Williams, L., 2003. Assessing test-driven development at IBM, 25th International Conference on Software Engineering, 2003. Proceedings, pp. 564–569. https://doi.org/10.1109/ICSE.2003.1201238

McCabe, T.J., 1976. A Complexity Measure. IIEEE Trans. Software Eng. SE-2, 308–320. https://doi.org/10.1109/TSE.1976.233837

Mekterović, I., Brkić, L., Milašinović, B., Baranović, M., 2020. Building a Comprehensive Automated Programming Assessment System. IEEE Access 8. https://doi.org/10.1109/ACCESS.2020.2990980

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M., 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey, Memon, A.M. (Ed.), Advances in Computers. Elsevier, pp. 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

Peffers, K., Tuunanen, T., Gengler, C.E., Rossi, M., Hui, W., 2020. The Design Science Research Process: A Model for Producing and Presenting Information Systems Research.

Pieterse, V., 2013. Automated Assessment of Programming Assignments. CSERC.

Rahman, K., Nordin, M.J., 2007. A Review on the Static Analysis Approach in the Automated Programming Assessment Systems.

Rößling, G., Joy, M., Moreno, A., Radenski, A., Malmi, L., Kerren, A., Naps, T., Ross, R.J., Clancy, M., Korhonen, A., Oechsle, R., Iturbide, J.Á.V., 2008. Enhancing learning management systems to better support computer science education. SIGCSE Bull. 40, 142–166. https://doi.org/10.1145/1473195.1473239

Semgrep, 2024. Semgrep [WWW Document]. URL https://semgrep.dev/about (accessed 6.23.24).

Thung, F., Lucia, Lo, D., Jiang, L., Rahman, F., Devanbu, P.T., 2015. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. Autom Softw Eng 22, 561–602. https://doi.org/10.1007/s10515-014-0169-8

Tikir, M.M., Hollingsworth, J.K., 2002. Efficient instrumentation for code coverage testing. SIGSOFT Softw. Eng. Notes 27, 86–96. https://doi.org/10.1145/566171.566186

Webster, J., Watson, R.T., 2002. Analyzing the Past to Prepare for the Future: Writing a Literature Review. MIS Quarterly 26, xiii–xxiii. https://www.jstor.org/stable/4132319

Virginia Tech, 2019. What is Web-CAT? [WWW Document]. URL https://web-cat.org/projects/Web-CAT/WhatIsWebCat.html (accessed 6.3.24).

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering, Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, pp. 1–10. https://doi.org/10.1145/2601248.2601268

Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., Di Penta, M., 2017. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. https://doi.org/10.1109/MSR.2017.2

Zampirolli, F.A., Borovina Josko, J.M., Venero, M.L.F., Kobayashi, G., Fraga, F.J., Goya, D., Savegnago, H.R., 2021. An experience of automated assessment in a large-scale introduction programming course. Computer Applications in Engineering Education 29. https://doi.org/10.1002/cae.22385

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., Vouk, M.A., 2006. On the value of static analysis for fault detection in software. IEEE Transactions on Software Engineering 32, 240–253. https://doi.org/10.1109/TSE.2006.38

Appendix 1. Survey questions

Course: Fundamentals of Software Testing (04.09.2023-20.10.2023)

Question 1: Motivation on this course was.

Type: Likert 1–5, where: (1) Very low, (5) Very High

Question 2: How much did you invest in your learning on this course?

Type: Closed

Options:

O1: I nether participated in the teaching events nor completed the required assignments

O2: I didn't participate actively in the teaching events, but I completed the required assignments.

O3: I somewhat participated and/or completed some of the assignments.

O4: I somewhat participated and/or completed all the assignments.

O5: I participated in almost all teaching events and/or completed the required assignments.

Question 3: The workload relative to the study credits awarded was.

Type: Likert 1–5, where: (1) Very light, (3) Appropriate, (5) Very heavy

Question 4: The teaching methods used on the course supported my learning.

Type: Likert 1–5, where: (1) Strongly disagree, (5) Strongly agree

Question 5: The course as a whole promoted my learning.

Type: Likert 1–5, where: (1) Very little, (5) Very much

Question 6: What aspects of the course need the most improvement?

Type: Open

Question 7: What was best about this course?

Type: Open

Question 8: How well have you understood the following topics?

Type: Likert 1–5, where: (1) Not at all, (5) Very well

Topics:

T1: Software testing V-model

T2: Testing in agile (Scrum) software development

T3: Alpha, beta, gamma, field testing

T4: Inspections

T5: Black-box testing

T6: White-box testing

T7: Grey-box testing

T8: A/B testing

T9: Test-driven development

T10: Unit testing

T11: Integration testing

T12: System testing

T13: Usability testing

T14: Acceptance testing

T15: Continuous integration and continuous deployment (CI/CD)

T16: Regression testing

T17: Bebugging

T18: Prototype testing

T19: xUnit

T20: Test documentation

T21: Test automation

T22: When to stop testing

T23: Test generation with Visual Studio Code

Question 9: How useful would you consider the following parts of the course?

Type: Likert 1–5, where: (1) Not useful, (5) Very useful

Topics:

T1: YouTube lecture videos

T2: YouTube demo videos

T3: Lecture slides

T4: Exercise events

T5: Exercise tasks

T6: Moodle news / texts

T7: Course project

T8: Exam

T9: Visiting lecture

T10: CodeGrade submission system

T11: Discord

T12: Wikipedia task

Question 10: What operating system were you mostly using when doing course tasks?

Type: Closed

Options: *Windows / MacOS / Linux / ChromeOS / Android / iOS*

Appendix 2. Survey data

Following tables display collected survey data as related to this thesis. Namely, data is presented for survey questions 3 and 9 as these questions were the primary consideration for the results of this thesis.

Question 3: The workload relative to the study credits awarded was.

Grade	Number	%
1	0	0.0
2	7	8.9
3	57	72.2
4	13	16.5
5	2	2.5
Total	***79***	***100.0***

Average grade: 3.1

Standard deviation: 0.6

Question 9: How useful would you consider the following parts of the course?

	1	2	3	4	5	Avg	Sd	N
YouTube lecture videos	1	3	8	22	45	4.4	0.9	79
YouTube demo videos	1	4	8	28	38	4.2	0.9	79
Lecture slides	0	4	11	28	36	4.2	0.9	79
Exercise events	8	9	34	15	11	3.2	1.1	77
Exercise tasks	0	2	8	31	38	4.3	0.8	79
Moodle news/texts	5	9	30	20	15	3.4	1.1	79
Course project	0	1	10	31	36	4.3	0.7	78
Exam	0	2	22	29	26	4.0	0.8	79
Visiting lecture	2	3	22	20	31	4.0	1.0	78
CodeGrade submission system	0	3	16	22	38	4.2	0.9	79
Discord	6	8	20	17	27	3.7	1.3	78
Wikipedia task	16	16	28	13	4	2.6	1.1	77