DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
68

EERIK MUULI

Automating the assessment and feedback processes in IT teaching – improving creation and maintenance from the teaching staff perspective

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
68

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
68

EERIK MUULI

Automating the assessment and feedback processes in IT teaching – improving creation and maintenance from the teaching staff perspective

UNIVERSITY OF TARTU
Press
1632

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on May 6, 2025 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

Assoc. Prof.	Marina Lepp University of Tartu Tartu, Estonia
Assoc. Prof.	Eno Tõnisson† University of Tartu Tartu, Estonia

Opponents

Prof.	Natalie Kiesler Technische Hochschule Nürnberg Georg Simon Ohm Nuremberg, Germany
Prof.	Lauri Malmi Aalto University Aalto, Finland

The public defense will take place on June 2, 2025 at 12:15 in Narva Rd. 18-1019.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (pdf)
ISBN 978-9916-27-880-2 (print) ISBN 978-9916-27-881-9 (pdf)

University of Tartu Press
http://www.tyk.ee/

To my family and friends

ABSTRACT

The Information Technology (IT) field has been rapidly growing for the past few decades which has resulted in an increasing demand for skilled professionals. Such demand has led to a significant increase in the number of students pursuing IT-related education and careers. While this expansion is beneficial for the industry, it has created numerous challenges for the educational institutions offering IT education. These institutions have to manage increasing class sizes while maintaining high educational standards, for example, by providing timely and individualized feedback for students. Traditional methods of assessment and feedback are often becoming unsustainable since the workload of the teaching staff also increases due to the need to manage more courses, students and resources. These activities have become time-consuming and repetitive, and this time could potentially be better utilized elsewhere. This thesis addresses these challenges by exploring the potential for automation in assessment and feedback processes in IT education, with a focus on programming courses.

The research methodology involved a detailed analysis of the current challenges faced by educators at the University of Tartu in teaching and learning activities related to automated assessment and feedback. This thesis used a qualitative approach in the form of mini-group interviews and inductive content analysis in order to capture the perspectives of the teaching staff on these topics. These interviews provided valuable insights into the needs, expectations, and concerns of educators regarding automated assessment and feedback. Inductive content analysis was used to identify key themes and patterns, which highlighted the necessity for automation tools that simplify the creation and maintenance of automated assessments.

The analysis was followed by designing and implementing the proposed systems. One system developed as part of this research is capable of automatically assessing programming tasks with graphical output. This system uses image recognition to detect the specified objects from the graphical outputs generated by students' code. The image recognition service returns a score indicating the probability of the given object being present in the image, and the system makes an assessment based on this score. The system was evaluated in multiple programming courses at the University of Tartu, where it received positive feedback from participants, particularly for the instant feedback it provided. This system significantly reduced the manual effort required from instructors and provided timely assessments of graphical programming assignments which had previously been a manual and time-consuming process. Additionally, this thesis introduced another system to standardize and simplify the creation and maintenance of automated assessments. The system consists of the Test Specific Language (TSL), its parser and compiler, the Tiivad assessment library, and a user interface that orchestrates these components. This TSL-based system enables the definition of assessments in a language-agnostic and structured manner while focusing on easier develop-

ment, maintenance, and modification of assessments without requiring extensive technical expertise. The integration of the TSL framework supports both dynamic and static assessment tasks, from basic syntax checks to more advanced functional tests, including object-oriented concepts. By focusing on a user-friendly interface, this system significantly reduces the complexity involved in creating and maintaining automated assessments. This also lowers the barrier to using automated assessment across various educational settings. The system was used and evaluated in a programming course and received positive feedback from users for its ease of use and valuable time-savings. The results of this thesis offer beneficial insights for IT educators in different educational institutions with a focus on teaching programming. More specifically, the results provide value in programming courses with a large number of participants. These systems help educators to design and automatically assess a wide range of programming assignments, including tasks with graphical output, in a user-friendly approach. The developed systems and gathered insights in this work can serve as a framework for future research and development.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ORIGINAL PUBLICATIONS

Publications included in the thesis

I Muuli, E., Lepp, M., Palm, R., & Luik, P. (2021). Automation of assessment and feedback in IT teaching from the teaching staff perspective. *2021 IEEE Frontiers in Education Conference (FIE)*, 1–9.
https://doi.org/10.1109/FIE49875.2021.9637290

II Muuli, E., Tõnisson, E., Lepp, M., Luik, P., Palts, T., Suviste, R., Papli, K., & Säde, M. (2020). Using image recognition to automatically assess programming tasks with graphical output. *Education and Information Technologies, 25*(6), 5185–5203. https://doi.org/10.1007/s10639-020-10218-z

III Muuli, E., Palm, R., & Lepp, M. (2023). Simplifying the creation and maintenance of automated assessments of programming tasks via Test Specific Language. *Proceedings of the 2022 6th International Conference on Education and E-Learning*, 14–20. https://doi.org/10.1145/3578837.3578840

IV Muuli, E., Lepp, M., Palts, T., Papli, K., & Palm, R. (2024). Reflecting on Simplification of the Creation and Maintenance of Automated Assessments for Programming Tasks. *2024 IEEE Global Engineering Education Conference (EDUCON)*, 1–3. https://doi.org/10.1109/EDUCON60312.2024.10578575

As the first author of each publication, the author was in a leading role throughout the research process. Responsibilities included designing and implementing the research methodology, conducting data collection and analysis, and interpreting the results. The author also handled all technical aspects, such as coding and system development. The author received assistance from co-authors when needed, such as during the coding process in data analysis. Although valuable feedback and guidance were provided by co-authors and supervisors, the author was independently responsible for the research presented in these publications.

Other published work of the author

V Muuli, E., Tõnisson, E., Lepp, M., Palm, R., & Luik, P. (2021). Automation of IT faculty work from the teaching staff perspective. *INTED2021 Proceedings*, 2839–2848. 15th International Technology, Education and Development Conference. https://doi.org/10.21125/inted.2021.0605

VI Muuli, E., Papli, K., Tõnisson, E., Lepp, M., Palts, T., Suviste, R., Säde, M., & Luik, P. (2017). Automatic Assessment of Programming Assignments Using Image Recognition. In É. Lavoué, H. Drachsler, K. Verbert, J. Broisin, & M. Pérez-Sanagustín (Eds.), *Data Driven Approaches in Digital Education* (pp. 153–163). Springer International Publishing.
https://doi.org/10.1007/978-3-319-66610-5_12

1. INTRODUCTION

This chapter serves as a foundation for the thesis by presenting the research problem, focus, and contributions. It outlines the growing challenges in IT education, particularly with the need for automated assessment and feedback systems. The purpose of this chapter is to fit this thesis into the broader context of IT education.

1.1. Research problem

For the past few decades, the shortage of personnel in the field of Information Technology (IT) and computer science has seen a significant growth (Oehlhorn et al., 2019). The number of open positions has been increasing faster than the number of qualified candidates, resulting in more and more young people showing interest in the field, acquiring IT-related education, and pursuing IT-related careers (Eurostat, 2024; Mets, Viia, & Ruusa, 2024). Educational institutions, from kindergartens to online platforms and universities, including the University of Tartu, are trying to facilitate such remarkable growth of interest in the field by creating more opportunities and student places for enrolling in IT courses. Noteworthily, students of different races, genders, ages and (academic) backgrounds are applying and showing interest in the IT courses (Chen et al., 2021).

However, the increased interest in learning computer science and programming has created challenges for the IT faculties and teaching staff (Pettit & Prather, 2017). One of the biggest challenges is accommodating such a high number of enrolling students while ensuring the quality of education does not drop with the increasing number of students (Charitsis et al., 2022). The growing size of IT courses results in greater workload for the teachers due to the need to manage more courses, students and resources. Some of the most common and elementary teaching activities are grading, providing feedback, and assessing the solutions of different (programming) assignments (Charitsis et al., 2022). All the aforementioned activities have become time-consuming and repetitive, taking up valuable time that could otherwise be used elsewhere by the teaching staff.

In order to alleviate the aforementioned issues, automation could be used as a potential solution by simplifying and automating the different faculty work processes such as grading, feedbacking, assignment generation, attendance monitoring, and many others (Paiva et al., 2022). Furthermore, automated feedback generation promises zero-cost support and instant feedback for the students in need of assistance (Phothilimthana & Sridhara, 2017). Different automation approaches become even more beneficial and valuable in courses with a large number of students such as Massive Open Online Courses (MOOCs), where efficient and accurate assessment and feedback is crucial (Charitsis et al., 2022).

Computer science institutions have studied, used and implemented automated assessment tools (AATs) for decades already (Hollingsworth, 1960) and this seems to be the common process nowadays (Pettit & Prather, 2017). There are many

ways automation is utilized in the software industry, but those methods or forms of automation might not be suitable for the educational context (Fulcini et al., 2023). Teaching staff play an important role in the digitalization and automation of different educational processes, as their insights into learning and teaching can potentially be highly valuable (Wohlfart & Wagner, 2023). In order to ensure that automated systems are supporting the educational needs instead of harming them, it is crucial to gather the opinions of the teaching staff and one way to do it is through conducting interviews with them.

There is a wide variety of programming languages and programming assignments that can be automatically assessed in different ways. The most common form of automated assessment for a program's functionality is dynamic assessment where the program is executed and its result is compared against a sample solution (Paiva et al., 2022). However, not all programming assignments have a fixed output, and some are more creative, such as programming assignments with graphical output (Thornton et al., 2008). Graphical tasks have proven to be popular and useful among students (Koren, 2024; Pugnali et al., 2017) and manually assessing them is time-consuming, therefore one specific area of interest is exploring and implementing such assessment tools to automatically assess programming assignments with graphical output (Douce et al., 2005). This type of assignments poses unique challenges since, in addition to the textual output of the program, the visual output must also be assessed. In order to address this, specific automation tools must be developed that can also work with the graphical output of the submitted solutions. The literature on automated assessment has described various methods and tools for assessing textual programming assignments (Messer et al., 2024), but there seems to be a gap when it comes to assessing assignments with graphical output: while there are some systems capable of assessing programming assignments with graphical output (Mertz et al., 2008; Tisha et al., 2023), there is no approach yet that would take the student's submitted code, generate graphical output based on it, and automatically assess it in educational settings.

Many institutions are building automation tools for a specific need only, doing it quickly and without proper standards, which often results in issues with long-term scalability and maintainability (Pettit & Prather, 2017). Such a high complexity for onboarding automated assessment can become a stopper for educators who may not have the necessary technical background (Skalka et al., 2019). This gap potentially slows down the widespread usage of automated assessment in the educational settings. Without having a standardized and easier way to create and maintain automated assessments, the potential benefits of automated assessment cannot be realized. Therefore, there is a gap to be filled when it comes to a user-friendly system that would simplify the creation and maintenance of automated assessments.

1.2. Research focus

This research is focused on exploring and developing innovative approaches and systems for automating the assessment and feedback processes in IT teaching. The study aims to simplify and improve the work processes of the teaching staff by leveraging automation. It is considering the perspectives and requirements of the teaching staff. Firstly, the research focuses on gathering the perspectives and needs of the teaching staff regarding automated assessment systems. The opinions and vision of the teaching staff were collected through interviews. The interviews help to identify the challenges and opportunities faced by the teaching staff regarding automated assessment and feedback. After gathering and analyzing the data, the further research direction was set in regard to the improvements of automated assessment systems.

Secondly, the research focuses on addressing the challenges of automatically assessing programming tasks with graphical output. The previous solution in which the assessment was performed manually is used as a base and data from it for validation. Different techniques and methods are explored for providing automated assessment of graphical tasks. After selecting an approach, the system is implemented and evaluated by students.

Lastly, the research focuses on simplifying the creation and maintenance of automated assessments by creating a standardized way of doing it. Test Specific Language (TSL) is a language designed to provide a standardized, programming language-agnostic, user-friendly, reliable and scalable way to define all the details that are needed for automatically assessing a specific assignment at hand. In order to remove the dependency of creating TSL files manually, a user interface was developed. Two additional components (parser and compiler) were developed as part of this thesis to validate the correctness of the TSL and generate the actual automated assessment code. This code is processed by the Tiivad library, which was also developed as part of this thesis to handle the assessment and feedback for different programming assignments. This approach helps to reduce the workload of the teaching staff while providing consistency and reliability among different assignments and courses.

Generative AI (GenAI) has shown great potential in feedback generation, programming assistance, and adaptive learning. However, its applicability to the specific challenges in this research remains limited. The primary focus of this study is on the automation and simplification of the creation and maintenance of automated assessments, rather than directly on different feedback mechanisms. Although GenAI tools offer promising capabilities, their suitability for grading in automated assessment is still evolving. Nevertheless, this thesis discusses their role in education and assessment, acknowledging their potential for future advancements.

This research is conducted within the University of Tartu, specifically within the Chair of Programming Languages and Systems (CPLS). Automation has been

used in programming education for many decades already in CPLS. Existing tools such as the Virtual Programming Lab (VPL) (Rodríguez-del-Pino, 2012) have been widely used for automated assessment, particularly in programming courses and MOOCs. However, existing automated assessment systems still face challenges. These include difficulties in creation and maintenance, limited adaptability to different programming tasks, such as those involving graphical output. Insights from teaching experiences at the university directly motivated this research. They shaped its focus on improving the creation and maintenance of automated assessments to make them more adaptable, efficient, and applicable across a wider range of programming tasks.

1.3. Contributions of the thesis

This thesis makes three contributions to the theory and practice of automated assessment. Each contribution is based on answering two research questions.

Contribution 1: Gathering and understanding the teaching staff perspectives and needs in regard to automated assessment and automated feedback. Contribution 1 is based on Publication I. Unlike prior research, which has largely focused on the technical implementation of automated assessment systems, this study systematically captures the perspectives of the teaching staff. By doing so, it identifies specific pain points that are often overlooked in system design, such as the balance between automation and human involvement in assessment. The qualitative approach used in this study provides a deeper understanding of faculty concerns and needs, which can guide the development of more effective and user-friendly assessment tools.

- RQ.1.1 Which of the faculty work processes related to assessment could be automated according to the opinions of the teaching staff?
- RQ.1.2 How does the faculty staff describe and characterize effective automated feedback?

Contribution 2: Development and implementation of a system capable of automatically assessing programming tasks with graphical output and students' perspective on it. Contribution 2 is based on Publication II. Existing automated assessment tools primarily focus on text-based programming assignments, whereas this research extends automation to graphical assignments using image recognition. While some prior studies have explored graphical output assessment, this research improves reliability by integrating configurable probability thresholds and evaluating the system in real educational settings.

- RQ.2.1 How can image recognition be used in order to automatically assess programming tasks with graphical output or visual representations?
- RQ.2.2 How do course participants perceive the performance of the automated assessment of programming tasks with graphical output?

Contribution 3: Designing and implementing a system that hosts Test Specific Language (TSL) together with its parser, compiler, automated assessment library Tiivad and user interface, to standardize and simplify the process of creating and maintaining automated assessments. Contribution 3 is based on Publication III and Publication IV. Unlike previous approaches that often require extensive manual coding and technical expertise to create automated assessments, this research introduces TSL as a structured, programming-language-agnostic method for defining assessments. Additionally, a user-friendly interface was developed to reduce the barrier to entry for instructors, making the adoption of automated assessment more feasible in various educational settings. This contribution addresses the scalability and maintainability issues that have hindered the widespread use of automated assessment in programming courses.

- RQ.3.1 How can a system for automated assessment of programming tasks be created in order to simplify the creation and maintenance processes?
- RQ.3.2 What impact does the new system using TSL have on the workload of teaching staff in large-scale programming courses?

1.4. Structure of the Thesis

This chapter introduced the research problem, outlined the research focus, and presented the key contributions. It also provided an overview of the challenges in IT education related to automated assessment and feedback. The following chapters explore these topics in greater depth. Chapter 2 explores the historical development of automated assessment, various assessment methodologies, and key features of existing automated assessment platforms. This is followed by Chapter 3, which describes the study's context, detailing the educational environment, courses, and technological infrastructure. Faculty insights on automated assessment and feedback are presented in Chapter 4, summarizing perspectives gathered through interviews. Chapter 5 then discusses the development and evaluation of a system for automatically assessing programming tasks with graphical output. Chapter 6 introduces Test Specific Language (TSL), a standardized approach for simplifying automated assessment creation. The design, implementation, and evaluation of TSL are covered, emphasizing its impact on reducing instructor workload. The findings are analyzed in Chapter 7, discussing their implications for teaching staff, automated assessment systems, and programming education. Finally, Chapter 8 concludes the thesis by summarizing key contributions, acknowledging limitations, and suggesting directions for future research. Supplementary materials, including TSL documentation, interview data, and courses context, are provided in the appendices.

2. BACKGROUND

This chapter provides an overview of the historical and current landscape of automated assessment systems. First, it discusses how these systems have evolved over decades and their relevance in different educational contexts. Secondly, assessing program features both via static and dynamic code analysis is discussed. Thirdly, features and limitations of existing platforms are described. Last, but not least, it examines the various types of automated feedback, analyzing their correlation to learning outcomes and to reducing teaching workloads.

Although Automated Assessment (AA) and Automated Feedback (AF) are closely related, they serve different purposes. AA focuses on grading by evaluating student submissions using predefined criteria such as test cases, output matching, or static code analysis. Usually, it provides evaluations in the form of scores. AF, on the other hand, extends beyond grading by guiding students through the learning process with explanations, hints, and suggestions to improve their solutions. AA systems can include basic feedback, such as correctness indicators or output comparisons. However, AF is primarily formative, supporting students in gradually improving their understanding through repeated practice and feedback. This distinction is important in this thesis: Sections 2.1-2.3 focus on AA for correctness evaluation and grading, while Section 2.4 shifts toward AF strategies that enhance learning, including different feedback strategies. By treating these as separate concepts, the thesis highlights how AA ensures objective and consistent grading, while AF supports learning through instructional guidance.

2.1. History of automated assessment

To give an overview of the history, major advancements were chronologically divided into logical units. Such separation of assessment systems into three generations was introduced by (Douce et al., 2005), and more recently, in a not so strictly divided historical overview of the automated assessment by (Paiva et al., 2022). Different assessment methods, systems and their characteristics are noted in the following paragraphs and the listing of example tools in them is by no means comprehensive but rather an illustration.

2.1.1. 1960s and 1970s (1st generation)

In the earliest automated assessment system, presented by Hollingsworth (1960), students wrote their submissions in assembly language on punched cards. Those punched cards were then executed with a specific grader program. The grader was only able to assess whether the solution was correct or wrong. The system was physically big and came with multiple limitations, including the fact that it was not completely automated in reality and required human interaction to work, and the solutions had to be written in machine language (Hollingsworth, 1960).

The system may seem simple and robust by nowadays standards, but it made it possible to more than double the number of students in a class. Furthermore, even with such basic functionalities, the security concerns already arose, and it became evident that students could harm the grading systems intentionally.

Along with the rapid growth in programming systems, assessment systems have also become more sophisticated. The first major leap was the possibility to grade programs written not only in machine language but also in programming languages (Berry, 1966; Forsythe & Wirth, 1965; Naur, 1964). Some examples include, for example, a program developed by Forsythe and Wirth (Berry, 1966) that asks students to write a procedure with specific parameters and is then tested with multiple sets of parameters and comparing the procedure results against predefined values. Another example is a program called GRADER2 that was developed and used in Stanford for grading novice students' ALGOL programs by generating random input data and validating the results (Forsythe & Wirth, 1965). Peter Naur (1964) developed a grading system that assesses the efficiency and logical completeness of students' algorithms instead of their formal correctness. The aforementioned systems make use of a grading program that works as follows: (1) the submission is transformed into procedures and injected into grader, (2) these procedures are sequentially executed with test data (also embedded within the program), (3) the execution time can be monitored as an option, and (4) the correctness of tests is evaluated by comparing the outcomes of a learner's procedure with those of the solution.

2.1.2. 1980s and 1990s (2nd generation)

In the 1980s software became more manageable and programming languages more user-friendly. These advancements made it possible to move from complex, difficult to maintain systems to much simpler GUI tools and command-line utilities (Isaacson & Scott, 1989). Isaacson and Scott (1989) outlined a "script-based" method and created a command-line tool that accepts a directory containing the learner's submission along with a predefined set of test input and output files as its input. As the submitted programs were potentially executed on the instructor's machine, more (security) concerns arose, such as wiping the instructor's disk, load testing the assessment system on purpose, etc.

For countering the aforementioned issues, TRY was introduced (Reek, 1989). There were two main advancements compared to the previous approach. Firstly, TRY wraps the assessment part of the hard drive with a directory so that it would look like a root of the file system. This essentially blocks access to the outer hard drive for malicious activities. Secondly, TRY makes it possible to limit the number of submissions for learners to force them to think thoroughly about their solution before resubmitting.

Program correctness was not the only thing that could be assessed and measured. The possibility of assessing CPU time, evaluating metric scores related

to code complexity and style, was introduced by a new system called ASSYST (Jackson & Usher, 1997). A notable feature of ASSYST is its role as both an automated assessment system and a grading assistant, allowing for submission handling, report generation, and the assignment of weights to specific test aspects through its graphical user interface (GUI).

As programming courses became more popular and more students enrolled, while evaluation was based on assessing the programming assignments, there was a need and opportunity to make the assessment asynchronous by creating an online assessment system, called BOSS (Joy & Luck, 1998). Students could submit their homework via BOSS and as a response an e-mail receipt was sent to the student confirming the submission. Teachers and tutors could use it to access the submitted solutions for testing and marking. The system was originally designed for Unix operating systems and C programming assessments, but later underwent significant evolution, adopting Java as a teaching language. It was transformed into a two-part system: (1) a module capable of assessing Java GUI applications, and (2) a tutor grading and assignment management application.

Computer-based assessment was not the only innovation that computers and access to the internet brought to programming courses and education in general. Not only specific assignments and assessments but also the whole course management, administration and even content delivery could be moved online (Benford et al., 1995). For such a purpose, the Ceilidh system was developed, which made it possible to divide courses into units and exercises together with the possibility of distributing lecture notes and other materials to students. Interestingly, the system found widespread use and survived many technological advancements for more than a decade, only to be succeeded by CourseMarker (Higgins et al., 2003).

2.1.3. 2000s and 2010s (3rd generation)

The next big leap in automated assessment systems occurred thanks to or in correlation with the widespread adoption and increased stability of the internet worldwide. In the 2000s, it became more feasible and reasonable to implement web-based architectures for automated assessment systems (Cheang et al., 2003; Edwards & Perez-Quinones, 2008; Higgins et al., 2003; Joy et al., 2005; Leal & Silva, 2003; Morris, 2003). Essentially, it means that there is a client running in the web browser that communicates with external services such as databases and other services that contain, for example, the assessment logic and test data, etc.

Many of the older systems, such as Ceilidh (Benford et al., 1995) that evolved into CourseMaker (Higgins et al., 2003) and BOSS (Joy & Luck, 1998), adapted to and acknowledged the rise of the internet. On the other side of the spectrum were many systems that did not survive the internet adoption phase and were gradually replaced by newer systems. This can also be related to the fact that many technologies are not relevant anymore and the skills that were required or assessed by those systems have antiquated (Swiecki et al., 2022).

Until around 2010, many of the automated systems were language-agnostic (evaluation relied primarily on output comparison), focusing on the most popular programming languages: Java, C/C++, Python, and Pascal. Interestingly, many new domains of assessment have appeared, including GUI development (English, 2004; Gray & Higgins, 2006; Muuli et al., 2017; Thornton et al., 2008), database management (de Raadt et al., 2006; Ke et al., 2009), assembly programming (Lingling et al., 2008), web development (Fu et al., 2008; Hwang et al., 2008; Sztipanovits et al., 2008), concurrent programming (Oechsle & Barzen, 2007), automated testing (Allowatt & Edwards, 2005; Edwards, 2003), and diagramming (Higgins et al., 2002; Waugh et al., 2004). From the perspective of educators, there have been enhancements in automated assessment tools. Beyond generating assessment summaries, some tools now offer instructors dashboards featuring statistical representations of student activity, primarily centered around submission results (Edwards & Perez-Quinones, 2008; Leal & Silva, 2003).

While discussing the latest trends in technology and automation, it is important to note the relevance of machine learning and AI-based techniques for automatically assessing and providing feedback for programming tasks. Some of the more traditional, statistical-based, approaches include the following models: random forest (Lazar et al., 2017), support vendor machines (Verma et al., 2021), and ridge regression (Srikant & Aggarwal, 2014). More recent, neural network-based, approaches include the following models: convolutional neural networks (CNNs) (Gupta et al., 2019), recurrent neural networks (RNNs) (Bhatia et al., 2018), and long short-term memory (LSTM) models (Nabil et al., 2021).

However, there are persistent challenges that remain, including grading solely based on output and providing limited feedback, often supplemented by instructor-defined hints. One potential remedy is utilizing industry-standard automated testing frameworks like xUnit-based tools, offering both familiarization with future professional tools and more detailed assessment (Amelung et al., 2008; Spacco et al., 2006). Additionally, promising approaches involve static analysis, assessing code without execution, to evaluate functionality by extracting metrics from source code and detecting plagiarism (Ahtiainen et al., 2006; Ala-Mutka et al., 2004; Prechelt et al., 2002; Rahman & Nordin, 2007). Experimental methods, like comparing learner attempts using graph similarity with known solutions, further enhance assessment (Wang et al., 2007).

2.1.4. 2020s (4th generation)

Another generation of automated assessment and feedback tools has emerged in the 2020s, which is driven by advancements in Large Language Models (LLMs), artificial intelligence (AI), and specifically generative AI (GenAI) (Prather et al., 2023a). Unlike earlier rule-based or static assessment approaches, modern AI-driven tools leverage machine learning, natural language processing, and generative AI models to provide more context-aware, adaptive, and dynamic feedback to

students (Kwak et al., 2023; Sun et al., 2024). These tools are no longer limited to research prototypes, but have already been integrated into real courses and have demonstrated practical benefits in programming education (Prather et al., 2024; Raihan et al., 2025).

One of the most significant developments is the rise of AI-powered code assistants such as GitHub Copilot, ChatGPT, CodeAid, and CodeHelp, which provide real-time support for students learning to program (Denny et al., 2024; Prather et al., 2024). These systems go beyond traditional automated assessment, which primarily evaluates correctness, by offering explanations, debugging assistance, and coding hints (Chang et al., 2025; Denny et al., 2024). Studies have shown that such tools can significantly improve the learning experience by providing personalized feedback that adapts to students' skill levels and misconceptions (Chang et al., 2025; Sun et al., 2024). For instance, a recent study found that ChatGPT-3.5 provided personalized feedback in 89% of cases for a simple while loop assignment, offering code style improvements and alternative solutions, which helped stimulate creativity and critical thinking among students (Azaiz et al., 2023).

Unlike earlier test-case-based assessment systems, AI-powered solutions are capable of interpreting student intent, providing guidance even when code is partially correct or follows alternative solution strategies (Leinonen et al., 2023). In contrast to conventional automated grading systems that focus on binary correctness, LLM-driven models can assess stylistic, structural, and conceptual correctness, considering whether a student's approach aligns with best practices (Prather et al., 2023a). Furthermore, generative AI allows for dynamic exercise creation, adjusting the difficulty of assignments based on a student's progression and previous performance (Kwak et al., 2023).

Despite their promise, AI-driven assessment and feedback tools introduce new challenges to programming education (Becker et al., 2023). Over-reliance on AI-generated solutions has raised concerns that students may use these tools passively rather than engaging in problem-solving themselves (Chen et al., 2021; Prather et al., 2023b). Additionally, issues of academic integrity and AI-assisted plagiarism have become a growing concern, as many of these tools can generate fully functional code snippets that students can submit without modification (Denny et al., 2024). Another challenge is that while LLMs provide impressive responses, they lack pedagogical awareness, meaning their explanations may sometimes be misleading or not aligned with instructional goals (Wermelinger, 2023).

Moving forward, the integration of AI-powered assessment tools into programming courses will require careful consideration of best practices (Prather et al., 2023a). Researchers emphasize the need for AI literacy training, ensuring that students use these tools ethically and effectively rather than as mere shortcut solutions (Prather et al., 2023a). Moreover, educational institutions may need to develop custom AI models tailored for programming pedagogy, rather than relying on general-purpose LLMs that lack explicit educational design (Raihan et al., 2025). Lastly, future AI-enhanced learning environments should consider hybrid

assessment approaches, combining automated feedback with human oversight to maintain educational integrity and personalized learning (Prather et al., 2024).

2.2. Assessing different program features

2.2.1. Dynamic code analysis

In 1994, Kay, Scott, Isaacson, and Reek (1994) brought focus to the challenge of consistently and thoroughly grading students' programs, particularly when it comes to dynamic aspects. Even small programs can generate multiple execution paths, which makes manual assessment impractical. Automation offers a solution by executing the program at hand with different inputs and configurations (Annor et al., 2022). However, executing students' programs exposes the system to potential bugs or malicious features that could harm the system. For instance, programs might try to access any files on the system without permission or consume excessive system resources such as memory or CPU (Liu et al., 2021). Therefore, it is highly recommended to run automated assessment systems in an isolated environment, often called a sandbox (Karvandi et al., 2022).

Functionality. Assessing functionality is the cornerstone of grading programs, with the main goal being to ensure that a program meets the requirements specified in the task description at hand (Paiva et al., 2022). In the vast majority of cases, this involves executing the program on a set of test cases, and afterwards comparing the generated output to the expected results. However, such a strict validation might be problematic in educational settings where it is not recommended to fail the test due to minor differences to expected results, such as white-space characters, capitalization or other minor typos (Paiva et al., 2022). In order to alleviate the well-known problem, many approaches have been proposed, including partial grading with pattern matching (Liu et al., 2019), regular expression matching (Pieterse, 2013), and comparing output tokens instead of characters (Yu et al., 2017).

As functionality assessment is a critical component both in educational and industrial environments, it is understandable that there are many industry-standard testing tools, like the xUnit family (e.g., JUnit, CUnit, PyUnit), that are widely adopted in the educational settings as well, offering fine-grained evaluation at class, method, and statement levels (Paiva et al., 2022). Automated assessment tools, such as JPLAS (Funabiki et al., 2013) and STAGE (Pape et al., 2016), leverage these frameworks. Additionally, web testing tools like Selenium (Siochi & Hardy, 2015) and mobile testing frameworks like Appium[1] are applied to automatically evaluate the functionality of students' web pages and mobile apps.

Efficiency. The efficiency of computer programs can be measured with the following parameters: time or duration, memory usage, CPU usage, disk space usage, network usage, and power consumption (Ihantola, 2011). The simplest and

[1] https://appium.io

most popular measurement seems to be the time (Millsap, 2010). The efficiency is usually evaluated by measuring and monitoring different metrics while the program is executed with various test cases (Enstrom et al., 2011). As the measurements are dependent on the test cases, the results are relying on the quality of the test cases design and the accuracy of the model solution that the program results are compared against (Hansen & Ruuska, 2003). Furthermore, the results can be influenced by various program features such as input/output implementations (Ala-Mutka, 2005). The feedback and results can be provided in various forms such as plotting CPU time against input size, as in OpenCPS (M.-Y. Chen et al., 2006). Some other tools capable of measuring CPU time are, for example, Assyst (Jackson & Usher, 1997) and Online Judge (Cheang et al., 2003). Assyst is also capable of measuring efficiency by calculating the number of times each block or statement in the program is called and comparing the results against a model solution.

Test Coverage. It is also desirable to teach and assess the quality of test suites. The software industry is striving for the highest quality of software, yet the ongoing issue with software defects causes a notable amount of downtime (Ray et al., 2017). In order to alleviate the vast amount of bugs, software must be tested thoroughly. Regrettably, the process of software testing is often viewed as monotonous and uncreative work by the developers (Steinhöfel & Zeller, 2024). This has also led to a noticeable gap in the knowledge required for writing meaningful and sufficient tests. As the curricula of undergraduate computer science are already packed, there should be a shift towards integrating the testing process into software development itself (Chisăliţă-Creţu et al., 2021). This can be done by different assessment strategies, such as requiring students to submit test datasets alongside their assignments to ensure students create and independently test their programs (Chen, 2004; Edwards, 2003). Chen (2004) uses a set of faulty instructor programs, grading based on the number of identified issues, while Edwards (2003) assesses the validity and coverage of test datasets against problem specifications.

Special features. In many systems, dynamic assessment occurs sequentially for multiple test data sets, preventing evaluation during processing, yet tools like Quiver (Ellsworth et al., 2004) enable instructors to establish state persistence between test cases for chaining different tests. Language-specific implementation challenges, such as dynamic memory management in C++, are often difficult for students to grasp, but tools like the Tutnew library (Rintala, 2002) can assess these issues in real-time by overriding memory management methods, printing assessment results upon program completion and detecting serious memory management errors.

Software programs featuring graphics and graphical user interfaces are appealing to students due to their impressive visual results, but they are challenging for automatic testing systems (Douce et al., 2005). Therefore, detailed requirements and rules are mandatory for programming tasks with graphics. Some such sys-

tems have been developed, though, for example a system by English capable of assessing GUI programs using JEWL (English, 2004).

2.2.2. Static code analysis

Unlike dynamic code analysis, which usually requires execution of the program, static code analysis is performed without executing the code but rather taking a deeper look into the source code and its features (Chelf & Ebert, 2009), for example, by using Python's Abstract Syntax Trees (AST) (Hovemeyer et al., 2016). The main advantage of static code analysis is the ability to analyze source code without executing it, which helps to uncover potentially critical issues that would go unnoticed by dynamic analysis due to the limited test cases and specific runtime conditions (Novak et al., 2010). Static code analysis is used for checking style, finding errors without executing code, discovering vulnerabilities, finding inefficiencies, measuring code quality with different metrics, and detecting plagiarism (Thomson, 2021).

Style. Automatic analysis of computer programs by demanding correct syntax is often enforced by language compilers and interpreters. Compilers and IDEs like GCC[2], Clang[3], Microsoft's Visual Studio[4], and JetBrains' IntelliJ IDEA[5] provide capabilities that tackle issues such as unused variables, type conversions, and non-standard language features. In the 1980s, more research was aimed at code readability and maintainability through style. Many tools such as Ceilidh (Benford et al., 1995), Assyst (Jackson & Usher, 1997), PASS (Dromey & Ryan, 1993) and Style++ (Ala-Mutka et al., 2004) were developed in order to assess code readability and compliance with software quality standards. Additionally, tools like Checkstyle[6], PMD[7], FindBugs[8], SpotBugs[9] and SonarQube[10] are frequently integrated into automated assessment systems to automate the measurement of code quality. They focus on identifying common programming mistakes made by novice programmers, such as unused variables, empty catch blocks, and unnecessary object creation. These tools work on a variety of common programming languages and are maintained to date. The integration of such tools into any automated assessment system helps students to improve the overall quality and maintainability of their code (Messer et al., 2024).

Syntax errors. While the assessment of program functional errors is usually done by dynamic testing against test data, syntax errors which can also lead to

[2]https://gcc.gnu.org
[3]https://clang.llvm.org
[4]https://visualstudio.microsoft.com
[5]https://www.jetbrains.com/idea
[6]https://checkstyle.sourceforge.io
[7]https://pmd.github.io
[8]https://findbugs.sourceforge.net
[9]https://spotbugs.github.io
[10]https://www.sonarsource.com/products/sonarqube/

programs not working properly can be identified through static analysis without executing the code. Some of the more common syntax errors include missing semicolons, unresolved variables, and mismatched parentheses or braces (Denny et al., 2014). In order to help students understand and overcome those issues, different tools have been developed. For example, CodeWrite is a web-based tool that provides automated feedback on students' Java code submissions in order to help them understand and correct their syntax errors (Denny et al., 2014). Similar purpose is served by CAP (Code Analyzer for Pascal) self-assessment tool that provides detailed, user-friendly feedback on syntax, logic and style errors in Pascal programs (Schorsch, 1995). For C++ programs students can use the automated assessment tool Athene that also provides clearer compiler error messages in order to help students understand and fix their syntax mistakes (Pettit et al., 2017). While not designed explicitly for educational purposes, these tools could serve as automatic aids for educators and students to pinpoint potential program errors efficiently.

Plagiarism. As mentioned earlier, together with the rise of computer science and automated assessment, plagiarism and its detection grew hand-in-hand due to the fact that programs are essentially paragraphs of text and those are easy to copy (Đurić & Gašević, 2013). Many modern tools now offer built-in plagiarism checks or the ability to integrate specialized solutions. Most of those checks and solutions are done via static code analysis without requiring any code execution (Paris, 2003).

Nowadays, it is very common to integrate specialized plagiarism tools (Novak, 2016) into any automated assessment system, although many already have built-in plagiarism checks (Chen et al., 2017; Nunome et al., 2010; Yu et al., 2017). These tools employ three primary methods for detecting plagiarism in source code: structural, semantic, and behavioral. Structural methods compare code structures, from token sequences (Aiken, 2002; Anzai & Watanobe, 2019; Inoue & Wada, 2012) to advanced data structures like trees (Fu et al., 2017; Zhao et al., 2015). Semantic methods, on the other hand, assess the meaning of suspected plagiarized source code using dependence graphs (Chen et al., 2010; Silva et al., 2020), control flow graphs (Chae et al., 2013), call graphs (Prado et al., 2018) and latent semantic analysis (Cosma & Joy, 2012; Flores et al., 2015; Ullah et al., 2020). Behavioral methods for detecting plagiarism analyze the student's behavior when submitting their code (Huang et al., 2020). For example, the frequency and timing of the solutions can be measured and compared against peers. One way to measure this, proposed in (Huang et al., 2020), uses an SCD (code similarity concentration) feature that references the Gini coefficient, showing how evenly the similarities of a student's code are distributed among other students in order to detect patterns indicating potential plagiarism.

Software metrics. At some point, in addition to program correctness and execution, software metrics, such as complexity, size and coupling, began to draw research interest as significant evaluation criteria (Rees, 1982; Van Verth, 1985).

Software metrics, as the name suggests, are numeric measurements used to characterize computer programs, serving as a basis for program evaluation and comparison. Although those metrics are easily obtained automatically, their educational relevance must also be considered, because metrics should align with learning objectives and instructional needs (Nguyen et al., 2021). For instance, the significance of requiring students to submit programs with a specific count of lines, comments, branches or statements should be done with caution, as these measurements are often tied to program style and design (Ljubovic & Nosovic, 2012). There are some widely recognized and used metrics though, such as Halstead's metrics (Caulo et al., 2021; Halstead, 1977) which count attributes such as operators and operands, and McCabe's cyclomatic complexity (Graßl et al., 2021; McCabe, 1976) which assess program complexity through control structures. In order to measure size, the Lines of Code measure (LOC) has been used for decades already in order to assess and predict quality of the software (Fenton & Neil, 1999; Monteiro et al., 2021; Njoku et al., 2024). Code coupling measures how heavily a module or a class relies on other modules. One way to measure this is via the Coupling Between Objects (CBO) metric (Chidamber & Kemerer, 1994; Martins et al., 2021; Ozturk, 2022; Romano et al., 2022). Higher coupling can lead to difficulties in maintenance and error-proneness while making modifications (Qusef et al., 2011).

Design patterns. There are many common mistakes that novice and sometimes even advanced programmers make while designing their programs. One can think about design patterns as best practices for solving those issues (Zhu, 2012). Usually design patterns help to organize the code in a way that would make it reusable by following specific rules within the interaction of classes and objects (Zhu, 2012). Although code design is not always the most straightforward aspect to automatically assess, there have been some efforts, such as Dong et al.'s work (2008) on recognizing design patterns, and Taherkhani's research (2008) on identifying sorting algorithms through static analysis. Some existing automated assessment tools mainly address lower-level design issues (Saikkonen et al., 2001; Truong et al., 2004), more specifically check whether the solution's structure aligns with predefined structures, such as the presence of loops or recursion. Furthermore, a tool has been developed that recognizes common design patterns from either program code or UML design specifications, which could aid in verifying that a student's program implementation aligns with the design document or teacher's design pattern requirements (Antoniol et al., 2001).

In summary, static code assessment should definitely be encouraged as it complements the dynamic testing methods. It enhances teaching staff's ability to assess the quality, performance and reliability of students' programs. Moreover, static analysis together with synthesis and repair techniques can be combined with dynamic testing to provide deeper insights and hints into why tests fail. It plays an important role for students as well by enforcing computational, structural, design

and functional thinking. These characteristics make it a valuable tool in programming education and assessment.

2.3. Features of automated assessment platforms

Automated assessment platforms have become crucial in computer science education by giving students immediate and scalable feedback (Falkner et al., 2020). Making automated assessment useful and maintainable depends on multiple factors such as the programming language chosen for teaching (Pears et al., 2007). The selection of programming language is important, but it is not the only consideration. Strickroth and Striewe (2022) highlight the fragmentation in task-based grading and feedback systems. Many such systems are developed without standardization or re-usability. They emphasize the need for scalable and interoperable solutions to ensure long-term sustainability and wider applicability in programming education. Additionally, this selection also depends on the faculty history and preferences, industry needs and the programming language characteristics (Pears et al., 2007). For example, Java, C, and C++ are widely used, including in our university, but they are not ideal for beginners due to their complexity compared to, for example, Python (Close et al., 2000; Palumbo, 1990). Connecting an automated assessment platform with learning management systems such as Moodle (Alstes & Lindqvist, 2007; Gutiérrez et al., 2010; Jimenez-Gonzalez et al., 2008) makes the course management easier, but on the other hand such integrations make room for security concerns (Alstes & Lindqvist, 2007; Gutiérrez et al., 2010; Jimenez-Gonzalez et al., 2008; Skalka et al., 2019). Sometimes a specific configuration and setup is needed if the domain taught is very specific. For example, having parallelism in programming assignments can cause unreliability and difficulty identifying root causes that require specialized tools like mentioned in (Oechsle & Barzen, 2007). There are many web-based assessment solutions that require an external network connection which might not always be desirable or doable (Paiva et al., 2022). Furthermore, teaching and grading practices, such as resubmission policies and possibilities for (semi)manual assessment, can also alter the design of automated assessment (Edwards & Perez-Quinones, 2008; Malmi et al., 2005). Last but not least, having publicly available platforms makes it easier for them to be adopted on a wider scale, but security must be guaranteed for such platforms (Amelung et al., 2008; Gotel et al., 2007).

2.3.1. Programming language

Typically, the selection of a programming language is determined within a specific context, taking into account factors like faculty preferences, alignment with industry requirements, technical characteristics of the language, and accessibility of pertinent resources and tools (Pears et al., 2007). Although languages like Java, C, and C++ are widely used both in industry and education, there has been significant discussion (Baharum et al., 2020; Close et al., 2000; Leping et al., 2009)

regarding their appropriateness for educational purposes, particularly when introducing programming to beginners (Pears et al., 2007). It is worth noting that these languages were not originally developed with a primary focus on educational use. On the other hand, languages explicitly designed for educational purposes, such as Python, Logo, Eiffel and Pascal, arguably offer advantages over more complex languages due to issues like verbosity and notational overhead while providing more syntactic simplicity (Palumbo, 1990). Automated assessment platforms also benefit from simpler programming languages because it is easier to analyze and assess them, which makes it also easier to provide more specific feedback without making students overwhelmed by complex syntax and concepts (Barr & Guzdial, 2016). The decision to choose a programming language should be context-specific, aligning with course objectives and learning outcomes, and educators must consider the broader curriculum when making this choice, recognizing that there is no one-size-fits-all solution in computer science education (Böszörményi, 1998).

2.3.2. Learning management systems

The integration of automated assessment of programming assignments within the framework of Learning Management Systems (LMS) has attracted considerable attention. Notably, there are several automated assessment tools designed to support programming assignments within platforms like Moodle (Alstes & Lindqvist, 2007; Gutiérrez et al., 2010; Jimenez-Gonzalez et al., 2008), Sakai (Suleman, 2008), Cascade LMS (Helmick, 2007; Nordquist, 2007), and Plone (Amelung et al., 2006, 2008). One of the reasons for this integration is to avoid maintaining both of these systems and potentially duplicating course management functionalities (Danutama & Liem, 2013). Furthermore, such integration makes it easy to reuse course materials, assignments and also automated assessments. In addition, keeping the systems isolated while interchanging only the most relevant information such as grades improves the overall security (Roy et al., 2015). Usually, LMSs host different courses, which can make them vulnerable to security risks. Potentially malicious code executed on any of the courses could cause more harm on the LMS, therefore, isolating the LMS from the automated assessment system would increase the security (Danutama & Liem, 2013).

2.3.3. Special domains

In addition to programming languages, the different exercise topics also play a crucial role in the setup of automated assessment as some domains require a unique setup and specialized type of tests. Visual programming has demonstrated its importance and usability in introducing programming concepts across various educational levels (Kelleher & Pausch, 2005). Programming tasks with parallelism often raise challenges due to their unreliability and errors where the root cause is not easy to identify. Such challenges require specialized tools such as

ASAS (Automatic Software Assessment System) developed by Oechsle & Barzen (2007). For databases, common automated assessment is performed by output comparison via scripts that query for example an SQL server, retrieve results and compare them to expected results (Kleiner et al., 2013). Such configuration lacks security, feedback and simplicity (Abiteboul et al., 2005). A simpler approach would be using a model that calculates syntactic similarity between the answer and reference solution (Panni & Hoque, 2020). Automated assessment of web projects consists of not only source code validation but also testing of functional user interfaces within web browsers. This means simulating user interactions with web content and verifying respective changes to the browser's state. Frequently, such assessment requires web servers and databases, often with external accessibility through specific ports. Peveler et al. (2020) introduced a system that uses Docker and its containerization concept to run students' code and the requisite services while ensuring isolation.

2.3.4. Resubmissions

Errors are an integral part of practicing programming, enabling students to learn from them. Nevertheless, it is important to have some sort of control over the number of resubmissions to discourage mindless trial-and-error approaches (Malmi et al., 2005). Some of the strategies include, most trivially, limiting the number of submissions and feedback while introducing deadlines, although this may lead to student confusion and mistrust of automated assessment (Guerreiro & Georgouli, 2006). Other approaches involve imposing time penalties that increase after each failed attempt (Janhunen et al., 2004), creating parameterized, randomly assessed assignments (Brusilovsky & Sosnovsky, 2005), or employing competition-style deadlines to motivate students (Guerreiro & Georgouli, 2006), but this raises concerns about prioritizing speed over quality. Additionally, hybrid approaches like Marmoset (Spacco et al., 2006) offer both unlimited and limited submission options with public and release tests to balance thorough testing with critical thinking.

2.3.5. Possibility for manual assessment

Combining manual and automated assessment offers a balanced approach where teaching assistants (TAs) can provide additional feedback or override grades as needed (Souza et al., 2016). Manual assessment or feedback can range from enabling teachers to view student submissions through the assessment system (e.g., VPL (Rodríguez-del-Pino, 2012) in Moodle) to making it possible to simultaneously display the TA feedback and automated assessment feedback (e.g., Web-Cat) (Edwards & Perez-Quinones, 2008). Although some systems may support grade overrides or feedback by TAs, it is critical to ensure the grading mechanisms are transparent to prevent confusion among both educators and students (Prather et al., 2019).

2.3.6. Public availability

The limited availability of open-source or freely accessible systems is a concerning and disappointing trend in the field (Gotel et al., 2007). Many research papers mention the development of prototypes without providing access to the tools, or those tools are too complex for integration (Rößling et al., 2008; Souza et al., 2016). Furthermore, teachers are often too busy to take on the challenge of finding such tools, learning them and building their courses and teaching practices around those tools (Barra et al., 2020). This all makes the reusability and cooperability more difficult and rather encourages different researchers and institutes to build their own in-house solutions.

2.3.7. Security

Securing the automated assessment process is of critical importance as there is often actual code execution. There are several approaches for tackling the security issues. For example, sandboxing tools like Systrace (used in EduComponents) could be used to provide a secure and controlled environment to prevent potentially malicious or erroneous code from affecting the host machine (Amelung et al., 2008). Furthermore, Merry (2009) discusses the use of the Linux security module for sandboxing. Some systems (Chen et al., 2006) are using static analysis, such as regular expressions, to filter out malicious code. Another widespread approach is to move the grading process to the client-side within students' computers, as exemplified by Mailing It In (Sant, 2009) and E-Commerce virtual laboratory (Coffman & Weaver, 2010). There is a similar approach where, instead of running the assessment on students' hardware, programs are run on a server separate from the rest of the system (Amelung et al., 2008). In addition to security features, sandboxing also helps to measure different software metrics as discussed above by setting hardware limitations on the machines.

2.4. Automated feedback

Delivering instant and suitable feedback is a significant challenge for introductory programming courses with a growing number of students. Feedback and grades provide an honest reflection on the students' learning and progress. Providing timely and adequate feedback becomes pivotal for students' academic progress. The most efficient and direct feedback coupled with assistance is in the form of face-to-face meetings with instructors (Chickering & Gamson, 2006; Wut & Xu, 2021). Providing such a "service" to hundreds of students coupled with feedbacking, grading and assessment quickly becomes repetitive and occupies an irrational portion of teachers' time (Phothilimthana & Sridhara, 2017). Some sort of automation is recommended in order to alleviate the load of the instructors and to make the supporting activities more scalable (Falkner et al., 2020). Providing feedback and grades automatically is an instant and cost-free option for supporting

students who need assistance (Phothilimthana & Sridhara, 2017). The submitted programs can fail during the compilation of the code or during the execution of the program (K. M. Ala-Mutka, 2005). However, running an error-free program is not enough, since the provided code must also give the correct answer (Paiva et al., 2022). In all the aforementioned failure points, the student must receive a corresponding error message and feedback (Sharma et al., 2018). It is important to note that automated feedback can be provided not only to students but to the instructors as well (Souza et al., 2016). Feedback can be provided on different aspects of the solution and in different forms (Messer et al., 2024).

Feedback strategies can be categorized into several types based on their cognitive and functional characteristics, as outlined by Narciss (2008) and Keuning et al. (2018). To provide structure, the different types of feedback are explored in the upcoming section. Three of the simpler feedback concepts are described in the following paragraphs.

Knowledge of Performance (KP). Running an error-free program is not enough, since the provided code must also give the correct answer (Paiva et al., 2022). KP feedback reflects on the overall performance level, such as the percentage of tasks completed correctly. For instance, "You completed 80% of the tasks successfully."

Knowledge of Results/Response (KR). KR focuses on the widely used binary assessment approach, which categorizes solutions only as correct or incorrect with nothing in between (Parihar et al., 2017). Such feedback is not optimal as partially correct solutions should also receive feedback and credits ranging, for instance, between 0 and 1 points.

Knowledge of Correct Results (KCR). KCR provides the correct solution or a model response, allowing students to compare their work against an ideal output (Keuning et al., 2018). For example, in all failure points, the student must receive a corresponding error message and feedback to understand their errors (Sharma et al., 2018).

More elaborated feedback concepts combine multiple types of insights and are detailed in the upcoming paragraphs.

Knowledge about Task Constraints (KTC). KTC offers hints on task requirements or rules, helping students align their solutions with given specifications. Some examples of partial assessment and feedback have been proposed in (Bettini et al., 2004; Ellsworth et al., 2004) where reflection classes that enable calling methods by their signature are used to assess functions and methods of the solution in isolation instead of assessing the whole solution together.

Knowledge about Concepts (KC). Provides explanations or illustrative examples to reinforce understanding of the subject matter. Systems such as ELP (Truong et al., 2003) assess individual statements of the submitted solution, which helps learner to understand the individual programming concepts.

Knowledge about Mistakes (KM). Identifies specific errors, such as syntax or logical issues, with varying levels of detail. It is possible to help students understand errors in their code by pinpointing the errors and showing similar examples

of faulty and correct code snippets with student-friendly error descriptions and fix examples as, for example, in the GradeIT system (Parihar et al., 2017). Systems like HelpMeOut (Hartmann et al., 2010) and TRACER (Ahmed et al., 2018) offer suggestions for compilation error based on an errors database or by comparing the faulty submission against error-free versions.

Knowledge about How to Proceed (KH). Suggests next steps or procedural guidance to rectify errors or progress further. Both students and teachers have something to gain from highlighting (perhaps even with different colors) the specific erroneous line and the character of the student's faulty submission (Dong & Khandwala, 2019). This way it would be easier for instructors to provide additional feedback, if need be, and students would also see right away where the culprit lies (Hartmann et al., 2010). Clustering submissions with similar features and providing automated feedback on them is a popular technique. There are both manual approaches to creating clusters as well as doing it automatically by using machine learning and program analysis (Head et al., 2017; Piech et al., 2015; Sharma et al., 2018).

Knowledge about Metacognition (KMC). Encourages reflection on strategies and processes, fostering self-regulation and independent problem-solving. Ideally, every student should have a personal tutor for constant feedback and support, but this is recommended for cases where there are few resources available and a small number of students (Kristiansen et al., 2024). Realistically, for most of the cases it is not possible due to different limitations. One way to alleviate the problem is to analyze students during the solving process (Fu et al., 2017). This can be done for example by gathering log data, on top of which predictive models can be applied and results generated and displayed on a specific dashboard for teachers that shows a visual overview of the whole class with an option to see a specific student in a more detailed view (Diana et al., 2017).

A good learning platform should apply multiple feedback types improve the learning experience (Narciss, 2008; Keuning et al., 2018). One such example is Codecademy, which is an interactive tutorial platform that integrates various feedback mechanisms (Jeuring et al. 2022). For KR, Codecademy provides instant feedback on whether the submitted solution is correct or incorrect. Furthermore, for KCR, it highlights differences between the learner's submission and the expected solution. In order to make them clearer, visual highlights are often used to clarify errors. Additionally, for KM, the platform provides detailed explanations for compilation errors together with task-related hints to help learners in correcting their code. Last, but not least, for KTC, static hints are provided for each subgoal of the task at hand. These rules are predefined and are not depending on the learner's solution, which ensures consistency in the task requirements.

Automated feedback is a pivotal component in IT education, starting from basic error correction and extending to more complex learning processes. However, there are still multiple challenges, such as scalability and adaptability, to tackle.

Future efforts should focus on improving the personalization by, for example, exploring technologies like machine learning and AI.

3. CONTEXT OF THE STUDY

Over the past few decades, there has been a remarkable growth of interest in the Information Technology (IT) specialty, and the University of Tartu is no different in this sense. The ongoing shortage of IT workforce increases the demand for the specialty among young individuals. This also means an increased number of IT enthusiasts enrolling in different IT-related educational institutions, courses and other forms of education. The University of Tartu is accepting hundreds of students from different backgrounds annually to take part in the various IT related courses and curricula. Interestingly, it is not only IT students who are interested in the courses but also students from various other disciplines. The increasing number of students enrolling and participating in IT-related coursework brings more work for the faculty staff as they need to juggle different teaching activities within multiple courses with an increasing number of students.

In light of the aforementioned perspectives, the present research explores faculty work processes related to automated assessment and feedback within the Chair of Programming Languages and Systems (CPLS) at the University of Tartu. The CPLS at the University of Tartu is responsible for the following courses: Computer Programming, Object-oriented Programming, Databases, Algorithms and Data Structures, Automata, Languages and Compilers, among others. A detailed overview of these courses, including enrollment numbers, student demographics, teaching resources, course content, programming language, is provided in Appendix A. Unless specified otherwise in Appendix A, all courses are conducted in Estonian. Additionally, most courses follow a traditional grading system (A–F). However, two introductory courses — Introduction to Programming (in Estonian) and Introduction to Programming (in English) — use a pass/fail grading system instead.

The Virtual Programming Lab (VPL) (Rodríguez-del-Pino, 2012) plugin for Moodle has been one of the key components in automating many of the processes in the courses, such as the creation of programming assignments and automated tests. VPL has also been used in many of the MOOCs hosted by the CPLS. In addition to Moodle-based automated assessments, there is another system, called Lahendus, developed within the CPLS, which allows students to view courses, assignments and grades, and submit their solutions for automated assessment. Lahendus is also synchronizing data on students and their grades from Moodle.

At the University of Tartu, faculty members can select and manage educational tools as long as they comply with GDPR and legal regulations. University management and IT support do not control the adoption of new tools, but instructors are expected to take full responsibility for their operation. This flexibility has led to diverse tools being used across courses in CPLS. There are some systems, such as the university's official study information system (SIS), that are centrally regulated and must be used. Additionally, if a system requires licensed software, funding must be acquired through grants or other institutional resources. IT sup-

port services are available when faculty members require technical assistance or when specialized configurations are needed. This decentralized approach enabled the development and deployment of the automated assessment solutions in this thesis without requiring institutional approval.

The author's role in the development and implementation of the automated assessment system was strictly as a technical expert, responsible for the technical aspects of automation and system maintenance. The author did not teach, co-teach, or participate in the instructional process, ensuring that their role did not influence the course outcomes or student performance. This distinction is important in maintaining the objectivity of the research, as the results were derived independently of the author's involvement in teaching-related activities.

The old Automated Assessment system used in Moodle relied on the VPL plugin, which presented several usability and maintenance challenges for instructors. While a few experienced faculty members managed to use the system effectively, many IT educators found it too complex to operate efficiently. A significant issue was the management of automated assessments, which required frequent duplication of the assessment library code and corresponding assessment code whenever a new assignment was created. This duplication process introduced a high risk of errors, as any modifications or fixes needed to be manually updated across all exercises, making maintenance cumbersome and inefficient. Additionally, instructors often relied on trial and error to configure assessments, leading to inconsistencies and inefficiencies in the grading process. Many adjustments were made by adding quick fixes (hacks) to existing codebases. This resulted in fragmented, non-standardized solutions with no backward compatibility. Moreover, the system lacked version control or history tracking, making it difficult to monitor changes, roll back to previous versions, or collaborate effectively on assessment improvements. These limitations significantly hindered the scalability and maintainability of the system, ultimately reducing its usability for a broader range of instructors.

The CPLS at the University of Tartu has been organizing introductory programming MOOCs since 2014 (Lepp, Luik, Palts, Papli, Suviste, Säde, & Tõnisson, 2017). A detailed overview of these MOOCs and their content can be found in Appendix B. This thesis primarily focuses on the 8-week course Introduction to Programming, conducted in Estonian, which was designed mainly for adults with little to no programming experience and diverse motivations (Luik et al., 2018). The course uses both self- and automated assessment (Lepp, Luik, Palts, Papli, Suviste, Säde, Hollo, et al., 2017). The participants can get automated feedback via different channels such as self-assessment questions and automatic "troubleshooters" that answer the most common questions. In addition, automated assessment is performed on Moodle quizzes and on programming tasks (Lepp et al., 2018).

Both, the regular and graphical assignments and automated assessments are hosted in Moodle and assessments are created with the Moodle's VPL plugin. VPL allows users to define test cases for each task. Some test cases are, for example, comparing input to the expected program output with the possibility to

provide feedback messages for different outcomes of the test. When a student submits a solution of a specific task to Moodle, the submission is sent to the VPL execution server that is hosted on one of the university's virtual machines. Every time a submission is sent, a temporary sandbox within the virtual machine is created to ensure the security of the system (Karvandi et al., 2022).

4. FACULTY INSIGHTS ON AUTOMATED ASSESSMENT AND FEEDBACK

This chapter addresses the following research questions: RQ.1.1: "*Which of the faculty work processes related to assessment could be automated according to the opinions of the teaching staff?*" and RQ.1.2: "*How does the faculty staff describe and characterize effective automated feedback?*". The chapter relies on Publication I which describes the results gathered through interviews conducted in the University of Tartu regarding the automation of assessment and feedback in the IT educational setting from the teaching staff perspective. The theme of the interviews and interview questions were conceived with a view of getting answers to the research questions. The chapter gives a thorough overview of the contribution in order to understand the current challenges and possibilities for improvement associated with automated assessment and feedback.

4.1. Data collection and sample

For data collection, a qualitative approach was chosen in the form of mini-group interviews. The selection of the mini-group format was based on the recommendation from (Anderson, 1990), as it allows to explore and share the automation topic in great detail and depth. As the interviewees had valuable experiences to share, interviewing them in small groups made sure that everyone had a chance to speak up and contribute, unlike in larger groups where potentially some individuals might steal the show and others' opinions might get overlooked or discarded.

The interviewees for the interviews were chosen from within the Chair of Programming Languages and Systems. The selection was based on their prior experience with automation and first-hand experience with teaching IT-related courses. Table 1 provides an overview of the interviewees, more specifically their respective roles in the faculty, their genders (M = male, F = female), and their ages (the second number in parentheses). The interviews were carried out in December 2019. The interviewees were grouped based on the time they preferred to be interviewed, while also trying to match up the most compatible individuals.

The goal orientation of the interviews relied heavily on the research questions because that sets the focus for the research. Therefore, the interview questions (Appendix C) were formulated based on the research questions. The questions could be split into five different types defined by Krueger (1994) as follows: opening, introductory, key, transition, and closing questions. After the opening part of the interview, the interviewees were asked two introductory questions regarding some of the more common and routine activities they have to perform in their teaching roles and a reflection on their prior experience with automation if any. After the introductory questions, four key questions were asked to pinpoint specific work processes that should be automated for teachers and students. The last

Table 1. People who took part in the interviews and contributed to the research (with age and years of teaching experience).

Interview no.	Interviewees
1.	Teaching Assistant (F1, 33, 8) Associate Professor (F2, 68, 25)
2.	Teaching Assistant (M1, 34, 6) Lecturer (M2, 60, 26) Teaching Assistant (F1, 42, 1)
3.	Lecturer (M1, 67, 18) Teaching Assistant (F1, 35, 7) Research Fellow (M2, 34, 5)
4.	Professor (M1, 51, 25) Lecturer (M2, 45, 17)
5.	Lecturer (F1, 33, 9) Associate Professor (M1, 70, 46)
6.	Associate Professor (F1, 52, 18) Associate Professor (F2, 39, 15)
7.	Lecturer (M1, 43, 18)
8.	Teaching Assistant (M1, 26, 1) Associate Professor (M2, 39, 6)

two key questions invited the interviewees to envision an ideal study process and to forecast possible future trends in automation without any limitations. The interview concluded with a question about the interviewees' idea of their own potential roles in making the future vision possible.

In total 8 interviews were conducted, including a pilot interview that kicked off the interviewing process. The pilot session is important as it was used for validating the overall processes, timing, questions, interview format, etc. After the pilot interview the questions were polished wherever needed, for example, if they were not clear enough, required additional explanation or even follow-up questions if the interviewees were not able to provide a reasonable answer within reasonable time. The pilot interview lasted approximately 78 minutes and had two interviewees. Notably, the results gathered from the pilot session were also included in the analysis and results. All the interviews were conducted by two persons and the roles were split as follows: one of the persons was responsible for leading the interviews, which involved doing the introduction, asking questions and follow-up questions, keeping the discussion flowing, and also summarizing and concluding the interviews. The other person was more in a supportive role, taking care of summarizing the key points, jumping in whenever needed, and keeping an eye on the overall flow of the interview together with the timing and interview structure. Before the actual interview, an introductory part was carried out in which the format and objectives were presented. At the same time, a consent for recording was obtained from each of the interviewees while informing them that those record-

ings will not be published and that their anonymity is ensured. The recordings were used solely for getting all the details of the interviews by later transcribing them. Furthermore, the interviewees were assured that their ideas would not be used against them and that offering ideas and providing solutions would not make them responsible or obligated to implement any of the suggestions. The average length of the recordings was around 71 minutes, which is fairly similar to the pilot interview that gave the general impression. All the recordings were transcribed by the thesis author word-by-word, without including filler words (e.g. "um", "uh") repetitions, and irrelevant noises to produce a polished text. Non-verbal expressions including (long) pauses, laughter, sighs or anything else were not included in the text. In transcriptions, speakers were labeled in the following format to differentiate between multiple speakers: "Interviewer 1:" and "Interviewee 2:". Although, there were only a few such cases, standard language was selected in order to prioritize readability and clarity meaning that any non-standard speech or words get transcribed into standard language. There were no timestamps in the transcriptions that would indicate the time of any given sentence. The shortest interview had 2941 words, and the longest one had 5737 words while in total all the transcripts added up to 29962 words.

4.2. Data analysis

An inductive content analysis approach was chosen in order to demystify and make sense of the qualitative data (Elo & Kyngäs, 2008). The input for the analysis were the transcripts from all the eight mini-group interviews. The phenomenon under investigation was a perfect fit for inductive content analysis in which, ideally, the subject matter has not been studied before or there are no conclusive results (Elo & Kyngäs, 2008). The transcripts were read through multiple times in order to be familiar with the data and have all the context as guided by Elo and Kyngäs (2008). After familiarizing with the data, the analysis process took place consisting of open coding, category creation, and abstraction. The analysis was done based on the research questions as suggested by Mayring (2000), which helps to go in-depth in the data due to the fact that the transcripts get analyzed multiple times from different angles (research questions).

QCAmap[11], a web-based tool, was used for category formulation and coding, with researches guiding the process and the tool facilitating the technical execution. The distinction between human actions and the tool's accomplishments in QCAmap lies in their respective roles in the transcription analysis process. Researchers drive the analytical decisions by first uploading transcription(s) into the tool, highlighting relevant passages, and tagging them with appropriate codes based on their judgment and research objectives. They also define the categories for organizing the generated codes. The codes can then be organized via drag-

[11] https://www.qcamap.org/ui/en/home

and-drop functionality in QCAMap by moving the codes into their respective categories as the researchers see suitable. Additionally, human interpretation is essential for deriving insights and conclusions from the coded and categorized data. QCAmap serves as a facilitator and technical enabler by providing an interface for coding and categorization features. The tool also makes it easier to compare coding results across multiple coders.

Meaningful coding units were identified in the text as those parts of the text that conveyed an important overall meaning in the context of the research question (Elo & Kyngäs, 2008). These units ranged from short phrases to longer ones containing multiple sentences that remained meaningful and understandable on their own. For example, a shorter coding unit was "*useful on large-scale courses*", and a longer one, complaining about the assessment system, stated: "*It would be great if the automated assessment could accept multiple different solutions, because students might have different ideas on how to solve a specific task. Currently, however, each task allows for only one specific solution approach.*" The context unit for this analysis was the paragraph or sentence surrounding the respective coding unit.

The analysis yielded a total of 38 codes describing automated assessment (RQ.1.1) and 28 codes connected to automated feedback (RQ.1.2). The formation of the categories and subcategories for RQ.1.1 based on the codes can be seen on the Figure 1. Similarly, the relationships between categories, subcategories, and codes for RQ.1.2 are displayed on the Figure 2. After the initial coding phase, the next stage of the qualitative inductive content analysis involved grouping codes into categories and subcategories based on similarities in the content. This categorization was conducted by three different individuals in order to ensure a general understanding of the data and categories.

The first part of the coding was carried out in iterations solely by the author to establish internal consistency before involving a second rater. The coding was conducted on the entire dataset, meaning that the rater independently coded 100% of the interviews. After generating the first set of codes, the coding was repeated after a couple of months to identify discrepancies, potential new codes, and possibly irrelevant codes. This iterative process helped refine and stabilize the coding framework. The first iteration resulted in intra-rater reliability score of 70%. After the first iteration, another one was carried out, which concluded with an intra-rater reliability score of 85%. Such an iterative process ensures and increases reliability of the research. To further enhance reliability, another researcher independently coded the entire dataset after the author had completed the initial process. Initial inter-rater reliability was calculated at roughly 50%, which indicates significant differences between the raters. To resolve the differences, the codes were merged and unified through discussion until a common understanding was reached. After the harmonization, both researchers re-applied the codes independently which resulted in an improved inter-rater reliability of approximately 78%, which shows substantial agreement (Shweta, Bajpai, & Chaturvedi, 2015). Inter-rater reliabil-

ity was measured using the percentage agreement method, which determines the ratio of coding decisions where both raters reached a consensus (Gisev, Bell, & Chen, 2013).

4.3. Results and discussion

The following chapter describes and interprets the data gathered from the interviews. The aim is to understand which assessment processes could be automated and how effective automated feedback should look like from the faculty staff perspective. The discussion covers the different major topics and subtopics that became evident from the interviews. This analysis helps to identify the current challenges, benefits and potential improvements in the automated assessment and feedback from the teaching staff's point of view. Some quotations from the interviews are presented in the text, in which the parentheses after the quote specify the interview number together with the interviewee identification.

4.3.1. Regarding automated assessment

For automated assessment, the results from the analysis provided five main categories: Improvements of automated assessment, Use of automated assessment, Negative effect on students, Instructor-related topics, and Technical concerns (Figure 1). Some categories consists of subcategories providing more detailed and specific aspects of automated assessment. The definitions and anchor examples for the categories are presented in Appendix D.1 providing further clarification and illustrative evidence for each category. "Improvements of automated assessment" are focusing on how to improve the system by adding functionalities and considering more use cases. The "Use of automated assessment" category highlights different scenarios where automated assessment could be applied. Potential drawbacks of automated assessment are mentioned under "Negative effect on students", however, they are not discussed in depth in this thesis and are described in more detail in Publication I. Potential benefits for teachers while using automated assessment are described under the "Instructor-related topics" category. Last but not least, "Technical concerns" captures the limitations of automated assessment from a technical perspective. Some categories and subcategories related to automated assessment are described in the following paragraphs, but only those relevant to this thesis are included here. An exhaustive list can be found in Publication I.

Use of automated assessment.

Cases for use. The interviewees said that automated assessment is particularly useful in cases with a lot of students since it is highly scalable. An example of such usage would be MOOCs. This point has also been noted in previous research (Nafa et al., 2023).

The general consensus from the interviews was that a hybrid approach is the most ideal since it incorporates both automated assessment and human involve-

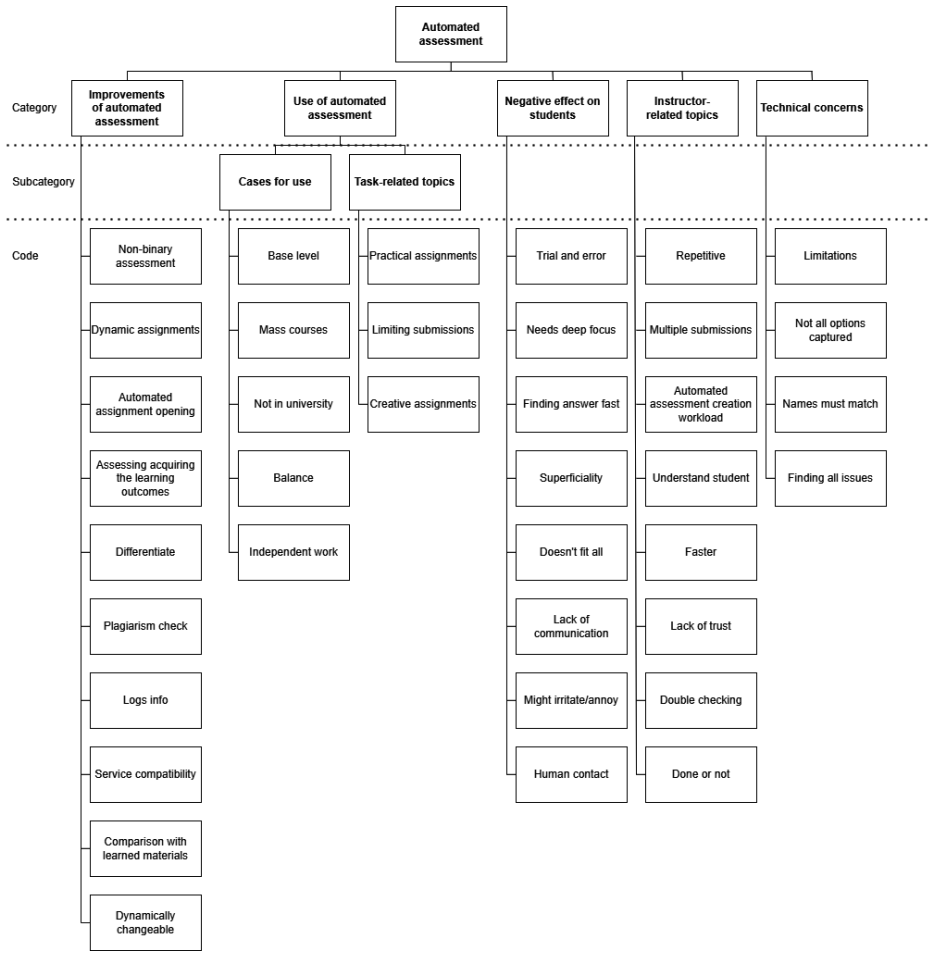

Figure 1. Categories, subcategories, and corresponding codes used for the formulation of automated assessment concepts.

ment. More specifically, it was suggested to use automated assessment for assignments that students perform at home so they can receive instant feedback. According to the interviewees, the key is to find a proper balance between automation and human interactions.

> *This automated check does help, but the human dimension is crucial. But the truth is likely somewhere in between. (2.M1)*

Task-related topics. The interviewees acknowledged the challenges of automatically assessing creative (graphical) assignments, but it is possible for some cases, for example, using JEWL to automatically assess GUI programs (English, 2004). The interviewees emphasized the necessity of assignments such as creating screen recordings and writing essays or texts with discussion. They also noted that students enjoy writing, but these tasks are time-consuming to assess manually, and it seems that automation does not provide a solution, either.

> *Furthermore, we have a task where screen videos need to be created. While each of them lasts only a couple of minutes, it takes a lot of time to review all of them in detail. If this process could be automated somehow, it would certainly be convenient. (6.F1)*

Instructor-related topics. Unfortunately, automated assessment currently comes with significant challenges. The interviews revealed that creating and maintaining automated assessments is a very complex and time-consuming process. Moreover, only a few individuals in the CPLS at the University of Tartu have the necessary skills and knowledge. The existing system is fragile and lacks proper documentation. Many interviewees emphasized the critical importance of creating a user-friendly interface or simplifying the process of creation and maintenance of automated assessments for the department. This issue is not unique to our institution and despite many skilled researchers working on automated assessment systems, most of them are designed for internal use and lack standardization and collaboration (Ala-Mutka, 2005; Paiva et al., 2022).

> *In addition, my thinking is that creation of automated checks could be simplified or perhaps improved somehow. So that it would be easy to create them and that I would not need to contact someone else whenever I need to change them. (2.F1)*

There seems to be a general consensus among the interviewees in the sense that automated assessment saves teachers' time, hence allowing them to focus on other aspects of teaching. However, teachers still tend to manually check the automated assessment results from time to time, indicating a trust issue in relation to automation. Some interviewees noted that if the output of automated assessment was a negative result, they would manually go over the results to make sure that the student did not lose any potential points. Ala-Mutka et al. (2004) have also noted that using automated assessment pressures instructors to thoroughly analyze and justify each assignment and test case.

> *Even when the test shows that everything is ok, I personally tend to make some random double checks. Did it really work how it was supposed to, did the student perhaps not invent something, which the automated check was unable to detect? (3.F1)*

Improvements of automated assessment. Although partial assessment of solutions is possible (Bettini et al., 2004; Ellsworth et al., 2004; Truong et al., 2003), the interviewees noted that the automated assessment system used in the CPLS is capable of providing only binary assessment, i.e., whether the solution was right or wrong. Such an all or nothing approach has its limitations and does not provide an accurate picture of students' performance. Many interviewees suggested assessing different components of an assignment separately and in different stages in order to support non-binary grading that would reflect students' knowledge more accurately.

> *Another helpful option could be, perhaps, splitting assignments into smaller parts. This would give us timely information not only on a single programming assignment but also on a specific part of the assignment. (5.M1)*

Technical concerns. The interviewees expressed concerns regarding automated assessment often being too tightly connected to the assignment related to it, which can cause limitations on the assignment. This means that making minor tweaks or changes in the assignment forces updates also to the automated assessment system. Additionally, for some instances, an assignment can have multiple correct solutions, but the automated assessment system may accept only one specific answer. For example, a programming assignment might expect a dictionary as a solution but should not force the dictionary keys to be in a specific order. Moreover, in some cases, the automated assessment system requires files to have specific names, and any submission with a different name can result in a score of zero points for the solution.

> *It would be nice if automated checks could accept a range of different solutions, because students tend to have different visions and ideas about how a particular task should be solved. At the moment, there is, in principle, only one course of solution envisaged for each task. (6.F2)*

The interviewees noted that one the worst scenarios is when the automated assessment system accepts a faulty submission or when students somehow manage to cheat the system in a way that they would receive points for submitting an incorrect solution.

> *With automated checks, it sometimes happens that they need some kind of fix, and a resourceful student is able to make a type of mistake that is not expected in automated assessment. (3.M1)*

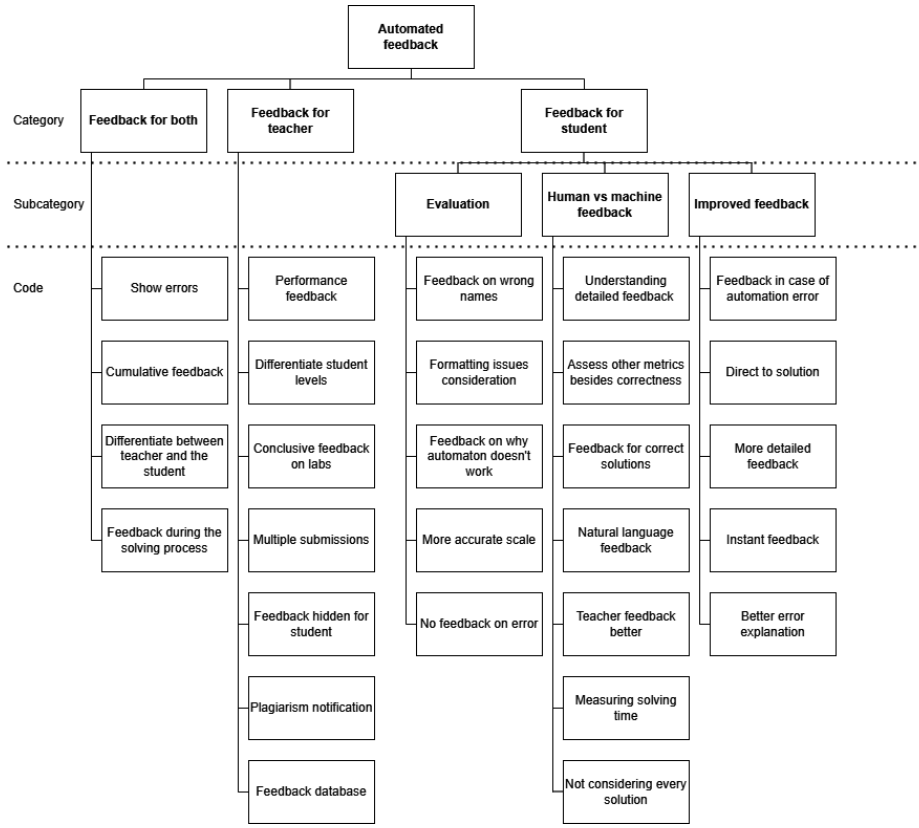

Figure 2. Categories, subcategories, and corresponding codes used for the formulation of automated feedback concepts.

4.3.2. Regarding automated feedback

The results from the qualitative inductive content analysis provided three main categories for automated feedback: Feedback for both (students and teachers), Feedback for teachers, and Feedback for students (Figure 2). Each category consists of multiple subcategories providing more detailed and specific aspects of automated feedback. The scheme shows clearly that the highest number of subcategories emerged about the feedback related to the students and the common feedback has the least number of subcategories. Such a distribution makes sense, since students are the main focus in educational processes, requiring the feedback received by them to be as detailed and optimal as possible. The definitions and anchor examples for the categories are presented in Appendix D.2 providing further clarification and illustrative evidence for each category. A few topics that are relevant for this thesis regarding students are provided in the following paragraphs. A more detailed overview with all the topics can be found in Publication I.

Feedback on the points awarded is often provided by the automated assessment system in a binary format meaning that the submission is reported as being either correct or incorrect. As per the interviews, this type of feedback is not very helpful

for students, since the solution can be partially correct and such a solution would receive at least a few points from instructors if assessed manually. Many interviewees recommended a non-binary assessment approach for automated feedback, in which even partially correct solutions would still receive some points for the parts of the assignment that are correct.

> *And also, that it would not simply pass or fail. That there would be points in between, for instance, 0.5 or 0.7. (6.F2)*

The interviewees suggested that, if possible, the feedback provided to the students should be as close to natural language as possible to make it similar to the feedback provided by the instructors. This means that the error messages should be detailed to provide students with as much support as possible in order for them to be able to understand why their solution is wrong and how to improve on it. Many interviewees noted that the students often do not understand why their solution failed the automated assessment or what is incorrect in their solution. The previously developed system GradeIT (Parihar et al., 2017) addresses the same issue by providing examples of both correct and incorrect code snippets along with modified error descriptions.

> *But the more intelligent the feedback systems we created, the better the feedback provided. When we look at the thousands of solutions and the feedback given in response to them, we can hopefully find a way to use them to synthesize feedback, based on machine learning. (4.M1)*

4.3.3. Recommendations for improving the current system

The interviews yielded several noteworthy findings regarding automated assessment and feedback. The ideas discussed above were compiled and illustrated in the following schema (Figure 3), which outlines the current automated assessment system together with its potential improvements and additions suggested by the interviewees. Those improvements are marked down next to the rectangular boxes in free form text.

First, generating assignments dynamically could reduce the risk of plagiarism, as each student would receive a unique variation of the assignment. In terms of automated assessment, the system should be able to evaluate different parts of a solution in isolation rather than using a binary approach (correct/wrong). Log data from students' solving processes could also be used to determine whether a student completed the assignment independently and to identify the most difficult parts where students tend to get stuck. Simplifying the process of creating and maintaining automated assessments is essential, and a more user-friendly interface would reduce the technical knowledge required from instructors. Additionally, running a plagiarism detection system automatically after students submit their solutions would help identify suspicious similarities and alert instructors accordingly. Error feedback should be improved by, for example, showing the exact

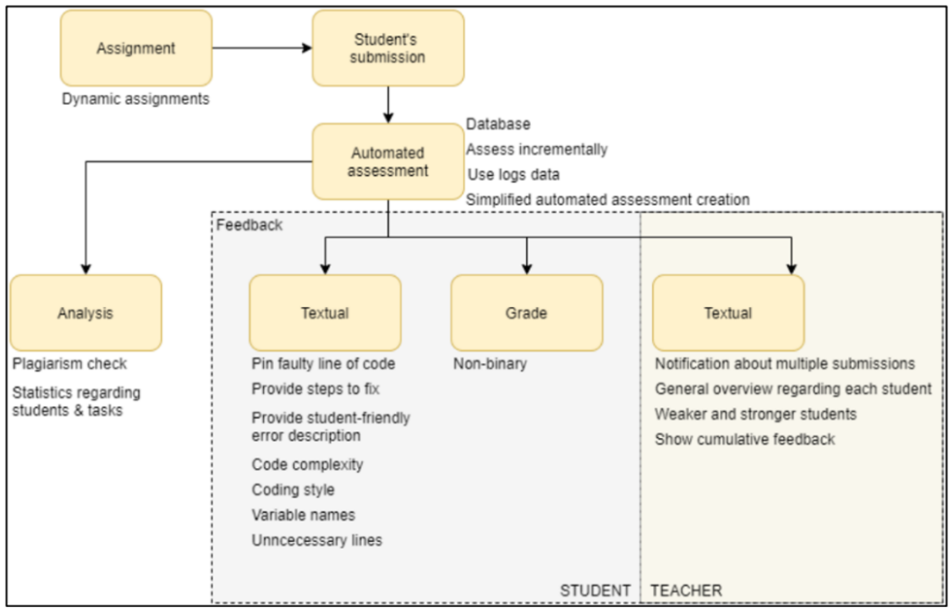

Figure 3. The schema of the automated assessment system used in the University of Tartu, with possible additions and improvements.

faulty line that caused the error and providing student-friendly explanations along with step-by-step solutions. The system should also detect and highlight issues related to code complexity, such as inefficient coding style, unnecessary lines, and poor variable usage, to encourage better programming practices. Various feedback tools for teachers should be developed, including dashboards that provide an overview of student progress, highlight struggling students, and track submission patterns. Furthermore, submission limit alerts could notify instructors when students submit tasks excessively, indicating a potential need for additional support. Finally, both students and teachers would benefit from cumulative feedback tracking, allowing them to review past assessments and feedback while monitoring their progress over time.

These insights are pivotal for identifying potential areas of improvement in the realm of automated assessment and feedback, contributing to the teaching and learning processes and academic experiences for both students and instructors. Overall, the interviews provided valuable insights into the current state of automated assessment in educational settings and offered actionable recommendations for improving these systems.

5. AUTOMATICALLY ASSESSING PROGRAMMING TASKS WITH GRAPHICAL OUTPUT

The following chapter focuses on the following research questions: RQ.2.1: "*How can image recognition be used in order to automatically assess programming tasks with graphical output or visual representations?*" and RQ.2.2: "*How do course participants perceive the performance of the automated assessment of programming tasks with graphical output?*". The research questions are answered and dealt with in Publication II, which proposed a solution capable of automatically assessing programming tasks with graphical output using image recognition. In this chapter we analyze the possibilities and challenges of automatically assessing the programming tasks that have graphical output, graphical representation or GUI elements. The chapter describes the implementation and provides insights and results gathered from the participants.

5.1. Graphical assignments

The weekly assignments in the course typically consist of four tasks and a quiz. The graphics-related tasks were introduced in the fourth week. Participants could choose an object they wished to draw programmatically, such as a flag, a traffic sign, or a house. Importantly, at least one of the tasks had to be submitted and passed in order to pass the course. Python's Tkinter library was used to program the graphical objects. One modification, introduced after the previous year's version of the course, was an additional task where participants could draw a freely chosen image simply by including a keyword for describing the object in Estonian language within their solution programs.

The first graphics-related task was to write code that would result in a graphical output containing a flag of an Estonian rural municipality (Appendix E). The passing criteria required that the flag needed to consist of at least three different colors or have an interesting shape (Figure 4). It was recommended to choose a cyclic flag (for example, waves on the flag) in order to use looping constructs in the solution code.

The second task was to draw a traffic sign. For this task there were no restrictions or limiting constraints, meaning that the participants could choose any traffic sign they wanted (Figure 5). However, it was still suggested to choose a cyclic traffic sign in order to incorporate looping constructs in the code.

The third task required drawing a house (Figure 6). There were multiple constraints on this topic. Firstly, the house had to contain at least three different elements such as a door, a window, a roof, or a chimney. Secondly, the house had to include at least three different colors. There were no restrictions on choosing the elements or colors.

Manually assessing over 1200 programming tasks with graphical output high-

Figure 4. Two participants' solutions to the flag task.

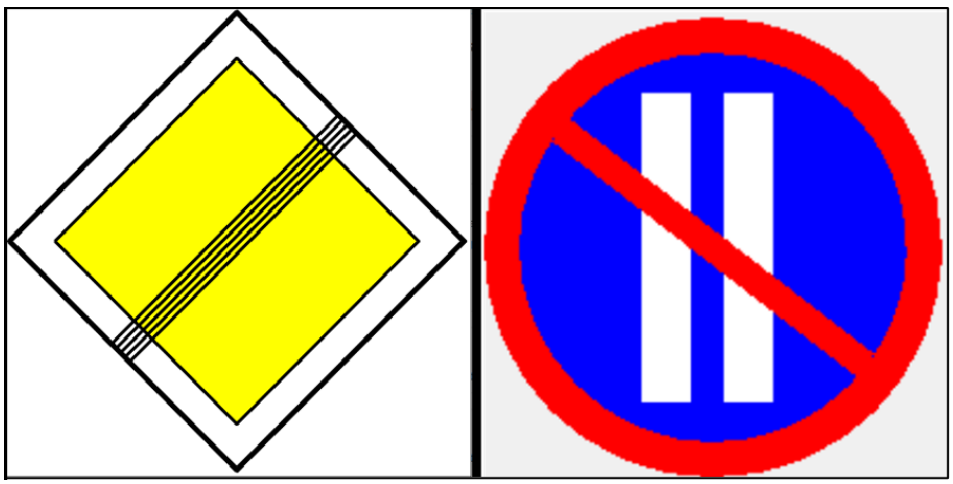

Figure 5. Two participants' solutions to the traffic sign task.

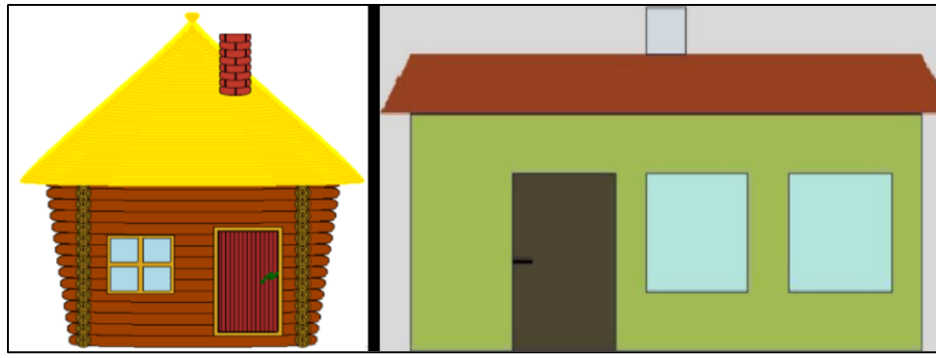

Figure 6. Two participants' solutions to the house task.

lighted the inefficiencies and challenges in the current process. Some of the challenges included bearing with very slow and congested Moodle forum threads for submissions and providing accurate and timely feedback. Automating the assessment became a necessary improvement.

5.2. Development process

Manually assessing tasks that involve graphical output is time-consuming, because in addition to the code, the generated output must be evaluated. In order to alleviate the workload from manually checking the graphical output, there are several image recognition service providers such as Clarifai[12], Google Cloud Vision API[13], and Imagga[14], to name a few.

The development of the new system followed these key steps (Muuli et al., 2017):

1. Analysis of the previous means of assessment.
2. Collection of the previous submissions.
3. Analysis of the image recognition service providers.
4. Implementation of the new system.
5. Testing of the new system on the previous submissions.

5.2.1. Analysis of the previous means of assessment

At first, the system for uploading the solutions and assessing them was analyzed. In the first course, there was a specific forum in which students had to upload their solutions into a dedicated forum created for it. It became evident that such a way was not sustainable for many reasons. First of all, there were more than 1200 solutions that had to be assessed manually. In case of faulty submissions, such as images submitted instead of the solution code or the code not working

[12] https://www.clarifai.com
[13] https://cloud.google.com/vision
[14] https://imagga.com

as expected, the author was informed about it. Furthermore, since the forum was accessible to all the participants of the course, everybody could see each other's work and not everyone actually wanted to share their code or artwork. In addition, the forum became very slow after hundreds of solutions had been uploaded.

5.2.2. Collection of the previous submissions

Since the first course had more than 1200 solutions it was reasonable to use these solutions as test data for the new system. A specific script was generated that downloaded all the solutions, including both the solution code and generated images if they were uploaded. If an image was not uploaded, it was generated from the student's code if the code worked as expected. The high number of sample codes and generated images was ideal for validating the stability and reliability of the new system in the later stages.

5.2.3. Analysis of the image recognition service providers

As opposed to regular novice programming tasks, in which the output of the solution code can be matched with the expected solution, the tasks with graphical output also require assessment of the generated graphical output. Since there is already a variety of services with the functionality of image recognition, a service provider capable of recognizing objects from provided images had to be chosen. There are multiple such service providers available on the market, such as Clarifai, Google Cloud Vision API, and Imagga, to name a few. In order to be selected, a service provider had to meet various criteria such as service speed, pricing, customer support availability, documentation, and capability to provide image recognition based on keyword. Clarifai was chosen as the suitable candidate since it was able to handle the necessary load when it was tested on previous year's solutions. Clarifai is known for its visual recognition capabilities since winning the ImageNet 2013 competition (Russakovsky et al., 2015). Furthermore, its free tier consisting of 5000 requests compared to 1000 for Google Cloud Vision did fit the course's requirements. Even though the customer support was unstable and, at times, misleading, they were able to provide valuable insights into their product together with the documentation in the end. Perhaps most importantly, it was possible to provide the image recognition API with keyword for finding specific objects in the presented image. Such functionality was not supported by the other service providers that were considered.

5.2.4. Implementation of the new system

The task's description says that the students have to use a Python library called Tkinter for creating the graphical output. As the first step, the submitted code is verified using Python's abstract syntax trees (AST) statically without executing the code (Hovemeyer et al., 2016). After verifying the correctness of the code, the submitted file is renamed in order to remove any special characters from the file

name that could cause issues. In order to generate the graphical output of the code, it has to be executed on a virtual machine. Since the Tkinter library requires a display and there is no real display on a virtual machine, it is simulated by installing a virtual display server called Xvfb[15] (X virtual framebuffer) that performs the graphical operations in virtual memory without showing screen output.

The students have to create programs with graphical output that is being displayed on the screen, but for automatically assessing the output, it also needs to be stored on the hard drive. In order to save the image, the submitted code is modified by injecting additional code that generates a PostScript (.ps) file. The PostScript file is then converted into a more common image format, .JPG, using a specific software program called GhostScript[16]. The GhostScript program is also installed in the virtual machine. After this process the final image is ready to be analyzed for finding specific objects.

The service responsible for recognizing objects in the image created from the submitted code is Clarifai. Clarifai's image recognition service is accessed via its API. The image created from the submitted code is sent to the API together with a keyword that describes the object that needs to be recognized in the image. For example, in the case of the assignment where a student needs to draw a traffic sign, the keyword "traffic sign" is sent to the API together with the generated image. The API expects a JSON payload containing the image, which is first encoded in base64, and also the keyword describing the object to be recognized. The response from the API is also in the JSON format containing a numeric value of the probability that the required object exists in the image provided. It was decided that submissions that received a probability score higher than 0.7 (70% probability) would be marked as "passed" and the others as "failed". A simplified schema of the system describing the process flow can be seen in Figure 7.

The probability threshold that determines the passing and failing of the test can be easily adjusted if need be. For example, the Clarifai service might be more accurate on some of the topics than on others. In case of a successful assessment, students received a message that their test passed. In case of a failure, the student was informed of their result, also mentioning the probability score that the Clarifai service scored on their image (Figure 8). The student was then advised to try to resubmit the solution or wait for manual assessment. The graphical tasks did not have a limit on the number of submissions, enabling students to improve their solutions and try different approaches.

In order to let students share their artwork, a special forum was created. The only rule, to avoid plagiarism, was that one could only see others' work after they had posted theirs first. Posting the graphical output of solutions was not mandatory for passing the course.

[15] https://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml
[16] https://www.ghostscript.com

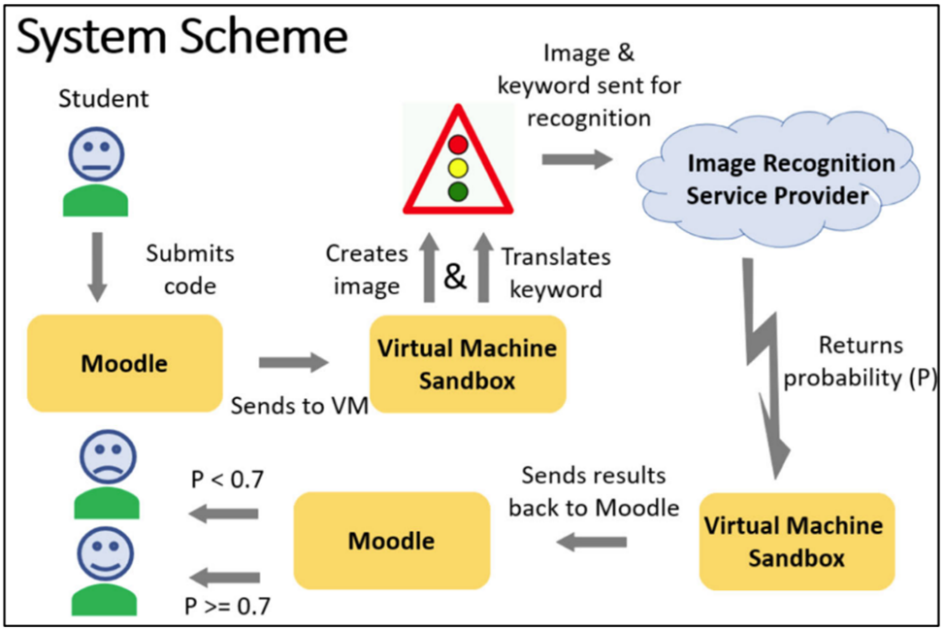

Figure 7. An illustrated and simplified schema of automated assessment for programming tasks with graphical output.

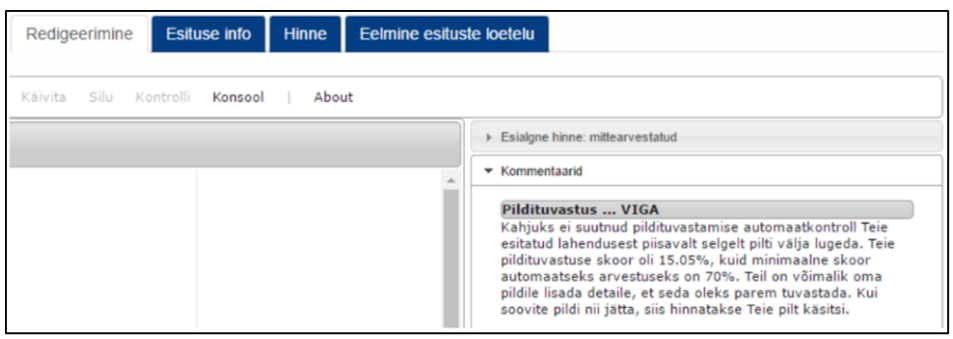

Figure 8. An example of a solution that did not receive the minimal probability score for passing the test. It received a probability score of 15.05% out of the minimal 70% to pass the test.

After one course, some improvements were added to the course and automated assessment. The most notable addition was the possibility to automatically assess programs with graphical output on any topic. For this purpose, a new assignment called “Drawing on a free topic” was introduced where students could choose their own topic they wanted to draw on (Appendix F). The only requirement for this test was that they had to add a single keyword as a Python’s comment on the first line of their program, stating the topic. What made the solution even more user-friendly was the possibility to write the keyword in Estonian instead of English. In addition to this, there were no limitations on the keyword selection, which meant that the participants could choose any word, including abstract terms. For example, if a student decided to draw a key, they could add a comment on the first line, such as “#võti”, which means “key” in Estonian. The system used Google Translator to translate the keyword into English to pass it on together with the image to the image recognition service provider Clarifai.

5.3. Research method

The MOOC “Introduction to Programming” (Appendix B) has been running continuously and the research focuses on three specific periods: March to May 2016 (Run 0), January to March 2017 (Run I), and October to December 2017 (Run II). Run 0, which had 1770 participants, is relevant for a comparison of manual and automated assessment. Run I included 1828 participants, out of which 989 (54%) successfully completed the course, and Run II had 1718 participants, out of which 1036 (60%) successfully completed the course. The results from Run I and Run II are presented separately due to significant differences between the runs.

The quantitative dominant mixed methods approach was used and therefore the questionnaires consisted of both closed and open-ended questions. Before the questionnaire was opened to the MOOC participants, it went through a piloting process. For the actual MOOC participants, completing the questionnaire was voluntary and did not affect their course progression. In both runs the questionnaire was administered shortly after the graphics-related tasks. In Run I and Run II, there were 766 and 915 participants, respectively, who answered the questionnaire.

The questionnaire in Run I included 14 questions, out of which some (Appendix G) were relevant for this research, and in Run II, there were six relevant items (Appendix H) in the questionnaire. The closed questions surveyed the participants’ opinions regarding the complexity and appeal of graphical tasks. During Run I, participants had to provide feedback on the "Draw a flag", "Draw a traffic sign", and "Draw a house" tasks separately. In Run II on the other hand, they were asked about the most challenging task overall among all the tasks assigned during the four weeks. Responses were measured on a 5-point Likert scale. Additionally, there were two open-ended questions that consisted of comments and descriptions of advantages and problems related to the general functionality of

automated assessment for graphical tasks.

For the closed questions, descriptive statistics were used to assess participants' opinions on the complexity of graphical tasks and the general implementation of automated assessment. The open-ended answers were qualitatively coded by the thesis author and another researcher and any differences were thoroughly discussed between them until a common ground was found. The coding for Run I resulted in inter-rater reliability score of 88% and for the Run II the inter-rater reliability score was 85% respectively, which shows an acceptable level of agreement (Shweta, Bajpai, & Chaturvedi, 2015). The inter-rater reliability was measured using the percentage agreement method, which calculates the proportion of coding decisions where both raters agreed (Gisev, Bell, & Chen, 2013). The analysis yielded a total of 20 codes describing Run I (Appendix I) and 23 codes describing Run II (Appendix J). For both runs, the coding resulted in identification of four themes: advantages and problems of graphical tasks, and advantages and problems of automated assessment of graphical assignments. The definitions and anchor examples for the categories are presented in Appendix D.3 and Appendix D.4, providing further clarification and illustrative evidence for each category.

In order to capture any false negatives and false positives, all the submissions from Run I were manually checked by the thesis author. Furthermore, the data from the system itself provided important insights. Specifically, insights were gained from analyzing the submissions that were either not automatically assessed or received a non-passing grade, as these cases helped to identify potential issues within the automated assessment system. There were 2272 submissions (from 1828 participants) for the graphical tasks that were automatically assessed in Run I. For Run II, the analysis was performed on the 23 failed automated assessments out of the 45 drawing tasks with the free topic.

5.4. Results

This section provides the findings from different phases of the system's evaluation. Results regarding the system's performance and participants' opinion on graphical tasks are provided. Lastly, various challenges such as keyword translation inaccuracies and keyword database shortages are examined.

5.4.1. Run I

The results show that the tasks with graphical output were moderately challenging, with an arithmetic mean difficulty rating of 3.4 (Figure 9). Notably, the task "Draw a flag" was submitted the highest number of times (570 out of 1118) among the respondents of this question despite almost half of the respondents rating its difficulty as 4 or 5.

Feedback on the graphical tasks showed that respondents liked the opportunity to choose the complexity level suitable for them, enjoyed comparing the artwork

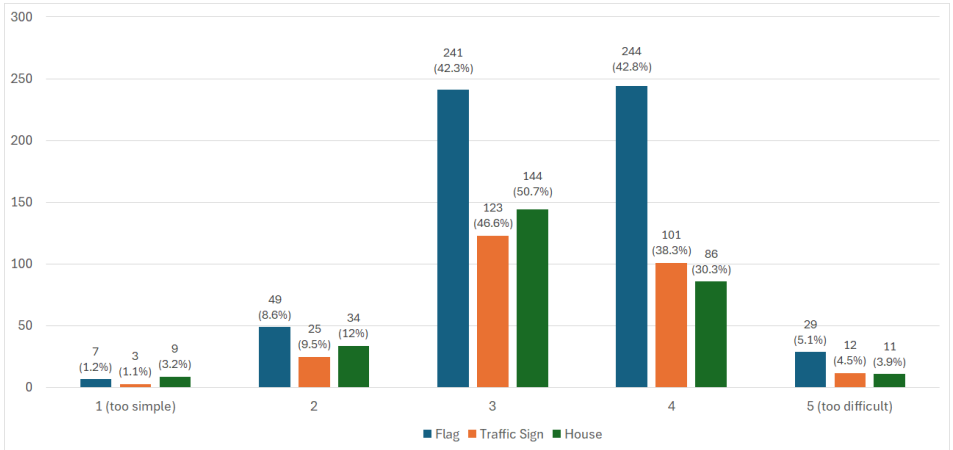

Figure 9. The results on respondents' perception of the difficulty of the graphical tasks.

generated amongst course members, but found drawing (programming) geometrical objects with coordinates somewhat challenging. Additionally, some respondents noted that they spent way more time on polishing their graphical tasks compared to other assignments. More than half of the respondents (404 out of 766) found the graphical tasks to be the most interesting out of the 8 tasks assigned in weeks three and four.

The general implementation of automated assessment for graphical tasks received positive feedback. It was rated on a scale from 1 to 5 where 1 means "does not work at all" and 5 means "works really well". The average score received for this was 4.4 out of 5. However, some issues were highlighted, such as situations where respondents had problems but still passed the assessment somehow, vague feedback from the automated system, and a lack of understanding about how the automated assessment works.

The analysis of open-ended questions for automated assessment of graphical tasks and graphical tasks themselves revealed key advantages and challenges. Automated assessment received praise for its efficiency, instant feedback, and reduced teacher workload. However, false positives, unclear failure reasons, and incomplete evaluations created issues of reliability, with some participants noting that they did not understand why their solution failed. Additionally, on some occasions, delayed feedback and penalization of non-standard solutions limited its effectiveness for complex or creative tasks. On the other hand, graphical tasks offered significant benefits by allowing choosing suitable complexity, encouraging creativity, and enabling real-time visualization of solutions.

Analysis of data produced by the automated assessment system revealed that 4.6% (104) of submissions contained false negative cases and 0.5% (11) had false positive grades. Some of the issues were easily mitigated, e.g., by addressing the threshold value for passing or failing test cases. Others on the other hand were more complex, such as the image recognition service not being able to find

the specified objects in the produced image. It is important to note that all the submissions were double checked manually for validation with specific emphasis on false positives, false negatives and on any issues that occurred.

Submissions that received a probability score higher than 0.7 were marked automatically as "passed". This threshold was chosen after testing the system on the previous submissions from Run 0 and based on the instructor's experience. The chosen passing criterion worked well, since the number of false positives was low. Although the threshold can be adjusted, it should be done with caution, because lowering it might increase the number of false positives.

During the first usage of the system (Run I), 28 hours of manual work were saved. That being said, the development of the system took at least twice the amount of hours saved. Importantly, the amount of hours saved will increase significantly with each of the following runs as little to no modifications will be needed in the system.

5.4.2. Run II

In Run II, the feedback regarding automated assessment was collected differently. More specifically, the feedback regarding complexity was gathered by asking about the most difficult task instead of separate ratings for each task. 45% of the respondents chose graphical tasks as the most complex ones, but the majority (46.1%) thought that the topics related to loops were the most difficult. 335 (36.6%) of the respondents marked the graphical tasks to be the most interesting ones amongst the tasks in the first four weeks of the course. The performance of the automated assessment of graphical tasks received very positive feedback also in Run II (Figure 10). There was also a free text comment section, which resulted in a similar conclusion to Run I. Additional positives from Run II included students enjoying the high availability of the assessment system and its ability to provide feedback on how to improve the solution. However, on the negative side, students pointed out that the system can be slow at times and lacked the capability to insert an additional image file to the drawing.

A new non-mandatory task for drawing on a free topic was introduced in Run II. It was tried out by 45 participants and the solutions outputted images on a variety of topics, including the following: laptop ("sülearvuti") (Figure 11), radiator ("radiaator") (Figure 12), car ("auto"), chess ("male"), ruler ("joonlaud"), cube ("kuubik"), heart ("süda") and egg ("muna"). Out of the 45 submissions, 22 (48.9%) passed the automated assessment and 23 (51.1%) were assessed manually for various reasons.

5.5. Challenges

One of the challenges was related to the automated translation of the keyword from Estonian to English. On some occasions, the keyword was translated inaccurately in a way that did not fit the context of the image. For example, in Figure

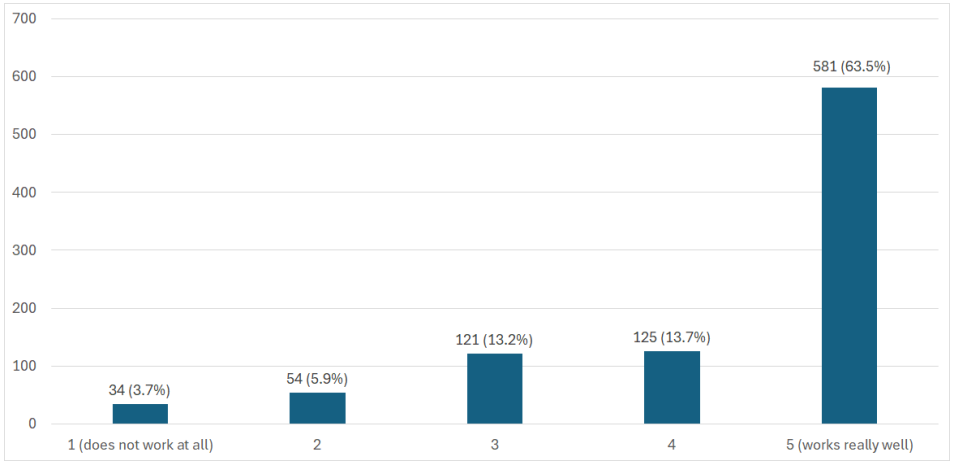

Figure 10. Respondents' evaluation of automated assessment of graphical tasks.

Figure 11. An example of a drawing of a laptop. The keyword used for translation was "#Sülearvuti".

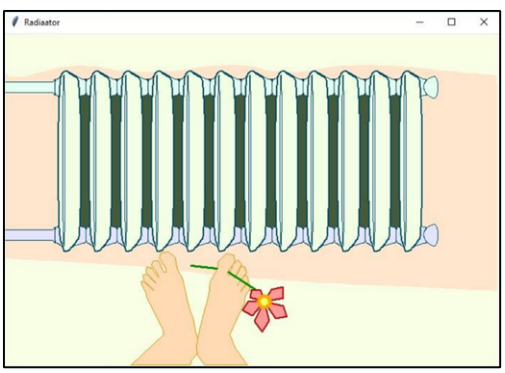

Figure 12. An example of a drawing of a radiator. The keyword used for translation was "#Radiaator".

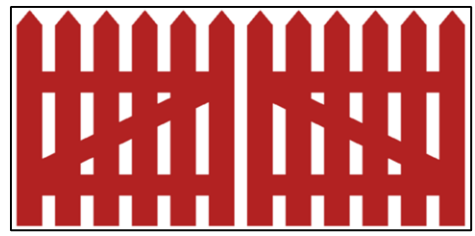

Figure 13. An example of a drawing of a gate. The keyword used for translation was "#Värav".

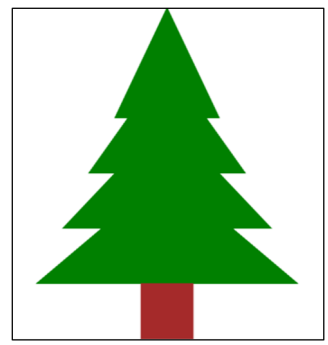

Figure 14. An example of a drawing of a spruce. The keyword used for translation was "#Kuusk".

13 we can see an image of a gate ("värav" in Estonian), but it was translated as "gateway". Such translation caused a very low probability score by the Clarifai service, because "gateway" is more often understood as something related to technology. If the translation would have been for example "fence", the submission would have received a high probability score, passing the test. Interestingly, changing the fence color to brown (instead of red) improved the probability score significantly for some reason.

Another example can be seen on Figure 14 containing a fir. The keyword used by the participant was "kuusk" which was automatically translated as "spruce". Interestingly, if the translation would have been "fir" the automated assessment still would not have passed it, but using a more general keyword such as "tree" would have passed the automated assessment. This leads to a suspicion that using more general keywords can provide better results than very specific keywords.

There were several keywords that did not have a translation (e.g., "Klaabu", a children's cartoon character) which resulted in a failed automated assessment. Furthermore, there were some keywords that were not present in the Clarifai's image recognition service keyword database, which also resulted in a failed automated assessment.

5.6. Conclusion and limitations

Several challenges must be overcome while developing such a system. The first challenge is identifying the necessary components: selecting suitable image recognition service and keyword translation and figuring out how to get access to various infrastructure components. The next step is configuring and integrating all the different components such as Moodle, the university's virtual machines, the assessment software, and the external service providers. Throughout this process, the stability and reliability of the whole system must remain unaffected since the infrastructure is also used for other services/courses. Also, the system must be stable and reliable for students to have a good studying experience. Given the positive feedback from the respondents and the time saved for instructors, the system will definitely be used in future courses.

There were some limitations on the image recognition service provider's side that became evident during the usage of the service. For example, the API expects only one keyword per image, although describing the context of the image can potentially be made easier by giving more keywords. Additionally, the vocabulary of allowed keywords was not publicly available. This resulted in some of the drawings not being automatically assessed, since the keyword provided was not recognized. The only way of knowing which keywords were allowed was through trial and error. Moreover, if the keyword provided in Estonian resulted in multiple translations in English, only the last one was used for recognition.

6. INTEGRATING TEST SPECIFIC LANGUAGE (TSL) INTO AN AUTOMATED ASSESSMENT SYSTEM

This chapter addresses the following research questions: RQ.3.1: "*How can a system for automated assessment of programming tasks be created in order to simplify the creation and maintenance processes?*", RQ.3.2: "*What impact does the new system using TSL have on the workload of teaching staff in large-scale programming courses?*". The research question is answered and dealt with in Publications III and IV. Contribution 3 discusses the development and implementation of a system that simplifies the creation and maintenance of automated assessments for programming tasks by utilizing Test Specific Language (TSL). It also shows the results and user feedback of the system in practice together with future directions. The chapter describes in detail the development processes, implementation and architecture details and concludes with the results from a piloting run as reported by multiple instructors.

6.1. Development process and system design

After the interviews described in Chapter 3 were carried out and analysis performed on top of the transcripts, the outcomes were shared within CPLS workshops as had been promised to the interviewees during the interviews. Such a format ensured transparency among the participants and also increased trust and interest in the topic. The goal of the workshops was to extract and prioritize actionable items based on the interview results in regard to the current automated assessment system (Muuli et al., 2021). As a result, it was concluded that the creation and maintenance of automated assessments for programming tasks should be simplified and a new system created for it. The new system should include, at a minimum, all the functionalities and capabilities currently available in the previous system. In addition, the new system has to contain some of the major functionalities that the teaching staff consider essential.

The first step was figuring out and designing the new system based on end-user preferences. It was concluded that different stakeholders such as teaching staff, students and potentially others would be using the system. The focus was set from the teaching staff perspective, though, as the process to be improved was the creation and maintenance of automated assessments. A process flow describing the teaching staff process of generating automated assessment descriptions in TSL via the user interface is depicted in Figure 15. The process starts with the teacher logging into the Lahendus system and selecting the desired course. The teacher must then decide whether the goal is to use an existing automated assessment or creating a new one. If the teacher decides to create a new automated assessment, they are directed to the page for creating a new automated assessment where they must fill out a form and confirm the creation. The system will automatically vali-

date the form for correctness and completeness. If the form is invalid, the teacher is prompted to fix the errors, which also get highlighted. If the assessment is successfully validated, the system generates the TSL file and links to the selected assignment. If the teacher decides to use an existing assessment, they can select it from the user interface and it will be added to the selected assignment without any additional steps necessary. Such a process improves the efficiency while reducing repetition. The diagram was later used as a basis for implementing the functionalities in the system.

Since there was no unified interface for describing and specifying automated assessment, it was decided to create a new declarative markup language called TSL (Test Specific Language). TSL should contain all the required information that a single automated assessment requires. This information includes inputs (both to functions and programs, also in the form of files), expected outputs (both for functions and programs, also in the form of files), passing criteria, and customizable messages for test outcomes. Initially TSL was written in XML that also contained a DTD (Document Type Definition) that described the structure of TSL in the form of elements and attributes, their order and nesting structure. After a little while it became apparent that maintaining, updating and documenting the XML format is not sustainable, since it is not very user-friendly nor readable. It was decided to migrate the TSL definitions to YAML format for improved readability and ease of use. In addition to YAML, support for writing TSL in JSON was also added for technical reasons. More specifically, since the system's backend programming language is Kotlin and we need to generate TSL from the user interface and vice versa, serialization of TSL is required. That being said, there was no built-in package in Kotlin that would work out of the box with serialization the way it was used. The easiest and most logical way to proceed was to use a functioning built-in Kotlin serializer[17] with JSON format, since it has a multiplatform support, conversion from YAML to JSON was painless, and this way the user interface and TSL interface could share a large part of the code. Currently, JSON is used as a default for TSL, but backwards compatibility is kept with YAML.

The potential end-users (CPLS teaching staff) were also kept nearby throughout the planning and development process. Weekly meetings were scheduled where new features were presented and reviewed, bugs were reported, and future ideas/directions presented. Keeping the end-users in the loop with the development process ensured the right direction by letting them shape and validate the features while keeping the user interface user-friendly and intuitive so that most of the teaching staff would feel at ease using it.

During the design phase of TSL, considerations were made regarding the test execution sequence. Initially, it was decided that the creator of TSL would have the flexibility to choose the order of tests. However, after listening to the feed-

[17] https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md

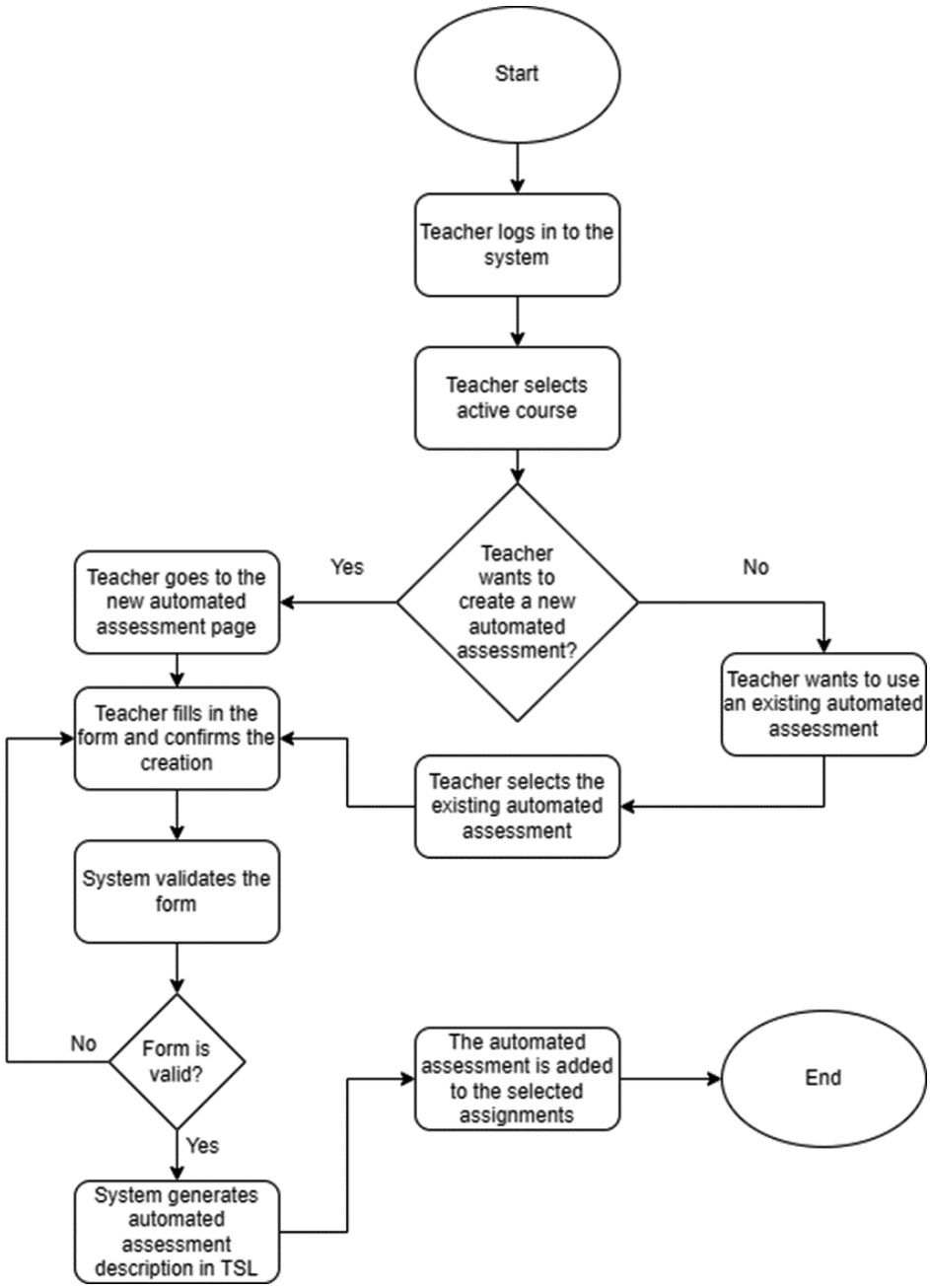

Figure 15. A process flow describing the teaching staff process of generating automated assessment descriptions in TSL via the user interface.

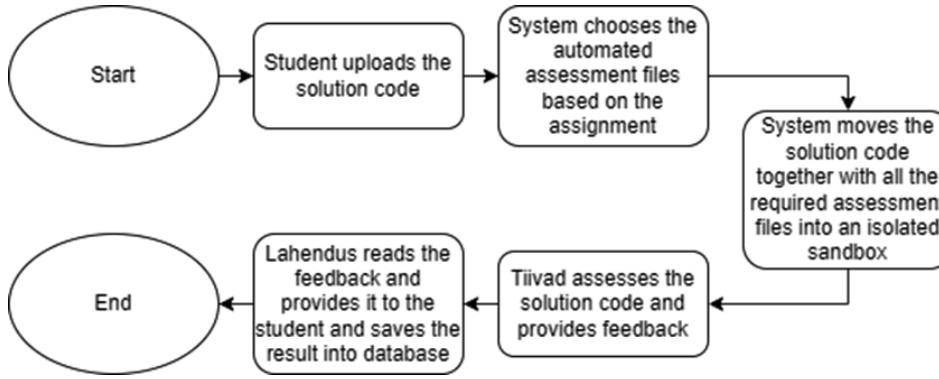

Figure 16. Process flow of assessing a submitted solution.

back from the end-users it became apparent that the sequence of tests did not significantly impact the assessment process. As a result, the option to select the test order was removed from the user interface, and the order was subsequently hard-coded to simplify the test creation and reduce room for error. Furthermore, the initial plan was that no other test would be carried out after a test failure was encountered. However, this was also modified during the testing phase in a way that all test cases would execute regardless of the outcomes of previous tests to provide more transparent and unified evaluation.

Parsing and compiling the TSL file required the development of a TSL parser and compiler to validate TSL correctness and generate required assessment code based on the TSL. Additionally, a Python library named Tiivad ("Wings" in English) was developed that executes the tests described in TSL and provides the results and feedback.

Once all the system requirements and components had been gathered, the next task was to integrate all the pieces and think about the actual assessment process flow in which the student uploads a submission that will get automatically assessed (Figure 16). The user interface was not available in the period when the development took place, therefore, separate documentation was created that described the creation and validation of a TSL file from scratch. This was the only way that the end-users were able to test out the latest changes and updates and provide feedback.

The final component of the system to be designed and implemented was the user interface (UI), as all the aforementioned functionalities and services needed to be in place beforehand. It was constantly reminded and brought to attention in the weekly meetings to ensure that the UI was intuitive for most of the teaching staff in programming courses. Initially, a mock-up was created (Figure 17) using Figma[18], that provided the first look into the UI's functionality and appearance.

[18]https://www.figma.com

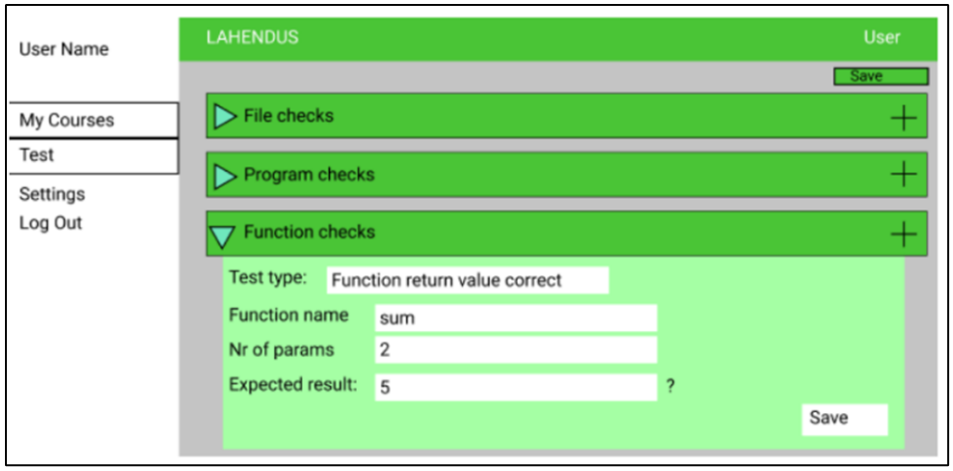

Figure 17. User interface mock-up generated with Figma.

6.2. Implementation

This section describes the practical steps taken to implement and integrate all the components within the automated assessment system. It outlines the core components developed within the thesis, including the TSL, user interface, parser, compiler, and Tiivad assessment library, and explains how they relate and communicate to each other in order to create a user-friendly solution. The implementation focuses on ease of use, scalability, and reliability to ensure that educators can define and maintain automated assessments in a user-friendly way without too much complexity.

6.2.1. Supported test types

Dynamic and static tests are widely used assessment types in software development. This system follows the same approach and it also includes the capabilities for testing some of the object-oriented programming (OOP) concepts (K.5). Each of these types of tests has its benefits and limitations and should be used for specific purpose. For example, we should not be using OOP tests if there are no OOP concepts within the assignment or solution.

In the current system, static tests are used for detecting specific characteristics within the student's code, such as the presence of loops, try-except blocks, or function calls (K.3, K.4, K.5).

In the created system, dynamic assessment is done by comparing its results against predefined expected outcomes. It is used for testing programs, functions and class instances within the scope of this research. At first, the student's submitted program, class, or function is executed, and afterwards different kinds of analysis are performed on them. For example, it is possible to validate if a specific function returned an expected value, whether a program generated expected output such as printing out expected strings, or if the program or function throws

an exception during runtime.

In order to extend the testing scope with more advanced programming concepts beyond basic programming constructs, OOP specific concepts like class definition, instantiation, and method invocation are used (K.5). This functionality was added since some of OOP principles are already included in beginner programming courses and automated assessment helps to ensure that students grasp fundamental OOP principles and their application.

By having the capabilities of assessing automatically both the fundamental programming concepts and more advanced concepts such as OOP principles, the assessment framework ensures a broad set of tools to assess students' programming skills. Combining the static and dynamic approaches in validating program, function and class behavior it is possible to catch a wide spectrum of issues, from regular syntax issues to class object initializations and much more.

6.2.2. TSL

TSL (Test Specific Language) is a configuration file written in either JSON or YAML format. TSL contains everything related to assessing a specific programming assignment at hand. TSL documentation describing all its functionality can be seen in Appendix K. There can be one or more tests, each consisting of one or more different checks. End-users can choose between manually creating or altering a TSL file or letting the system generate it for them based on the selections made via UI. TSL is currently focused on the Python programming language but does not set a hard requirement for it. It can be expanded without limitations to any other language/technology and integrated with any system. It is possible to utilize TSL with any existing automated assessment system by developing a custom parser and compiler coupled with an automated assessment backend service tailored for any specific requirements.

For one exercise, it is currently possible to test 1 to N different aspects of the code. The TSL file follows a certain structure (Figure 18). In the beginning, the programming language to be tested is defined together with the TSL version to be used. In addition, the file names to be tested are defined and a Boolean value that specifies if some preliminary tests are executed, e.g., testing whether the submitted file is empty or valid. After that, the different groups of tests have to be defined: program tests, function tests, or class tests. Each of these groups, in turn, includes individual tests. An individual test can have its own specific required and optional parameters. Some required parameters are test type and, for example, function name in case the test needs to test anything specific to a function. Some optional parameters are, for example, points weight to be given and messages displayed to the end user based on the test result.

Benefits of using TSL. The adoption of TSL for automated assessment of programming tasks offers multiple advantages over conventional methods such as manual creation of assessments separately for every assignment. TSL provides

a concrete structure and syntax for all the different types of automated assessment for programming tasks. Such an approach simplifies the creation of new automated assessments but also updating and maintaining the existing ones. The standardized approach brings consistency across all the different assessments. Every TSL file must follow a similar structural format, which also makes it easier to create, understand and maintain a larger number of assessments, since they are quicker to update and modify.

With TSL, the teaching staff do not need to worry about the underlying technical details and mechanisms that are actually performing the automated assessment. Instead, they can focus on creating and updating the instruction files that define what needs to be done and how. Such abstraction helps the teaching staff to put more effort in the pedagogical aspects of teaching rather than having to worry about the implementation details, potentially causing frustration.

By having a standardized format, the risk of human error is minimized. Of course, mistakes can still happen while defining the assessment logic and rules, but the mistakes that would happen while creating and updating the underlying implementation are eliminated. This leads to more reliable and robust assessment, which is important for fair evaluation.

TSL also makes it convenient to reuse assessment definitions across assignments and courses. It is easy to modify and adapt the TSL files for existing or new assignments, saving valuable time while ensuring the quality of assessment.

Although TSL is currently implemented primarily for Python assignments, it is designed to be language-agnostic. This means that it can be extended to support other programming languages in the future. This provides flexibility and scalability for different educational environments.

6.2.3. Automated assessment back-end called Tiivad

The actual assessment takes place inside the back-end via a dedicated Python library called Tiivad[19]. Tiivad was developed to take care of the automated assessment process. It takes as input the student's submission and the automated assessment script generated by the Compiler (see next subchapter). The generated script calls functions from the Tiivad library that takes care of the actual assessment. It then runs all the tests on the student's submission, generates feedback on each of them, and reports back results in a structured JSON format. The output is retrieved by the Lahendus system and reported back to the end-users (students and teachers). Tiivad is isolated from the rest of the system for security reasons as the student's code is executed and can contain malicious code. The assessment part is encapsulated inside a Docker container that is operated independently without accessing other parts of the system.

[19] https://github.com/emuuli/tsl-tiivad

```yaml
language: python3
validateFiles: true
tslVersion: 1.0.0
requiredFiles:
  - submission.py
tests:
  - type: program_execution_test
    id: 1
    standardInputData:
      - User input from keyboard
    inputFiles:
      - fileName: inputFile.txt
        fileContent: Content of file\nLine2\nLine3
    genericChecks:
      - checkType: ANY_OF_THESE
        id: 987
        expectedValue:
          - down
          - up
        beforeMessage: Verify if the program's output contains keyword.
        passedMessage: Program's output contains keyword.
        failedMessage: Program's output doesn't contain keyword.
    outputFileChecks:
      - fileName: outputFile.txt
        checkType: ALL_OF_THESE
        expectedValue:
          - Expected content of generated file
          - Line2
          - Line3
          - Line4
        beforeMessage: Verify if the program's output file contains keyword.
        passedMessage: Program's output file contains the keyword.
        failedMessage: Program's output file doesn't contain the keyword.
    exceptionCheck:
      mustNotThrowException: true
      beforeMessage: Verify that program doesn't throw an exception.
      passedMessage: Program didn't throw an exception.
      failedMessage: Program threw an exception.
    name: Program execution test.
```

Figure 18. A sample TSL file for a program execution test.

6.2.4. Parser & compiler

A parser[20] is used to validate the TSL file for any misconfigurations such as missing or extra fields and wrong field types. The parser reads in the TSL file as text, serializes its content and transforms it into a tree structure consisting of different test objects specified in TSL. After TSL has been parsed, it is passed on to the compiler[20] that extracts all the objects from the tree and converts them into a Python script consisting essentially of Tiivad back-end assessment function calls (Figure 19). For smooth integration of this new functionality into the existing system, the Kotlin programming language was used for developing both the parser and compiler as the existing system, Lahendus, is also developed in it.

6.2.5. System design

All the added/developed components, including TSL parser, TSL compiler, User Interface, and Docker container with Tiivad back-end, are facilitated in the Lahendus system. In the diagram (Figure 20) it is illustrated how the different components of the automated assessment system for programming tasks are structured and interact with each other.

Teachers can create and modify automated assessments for different programming assignments via the user interface. Once an assessment is created or modified and saved, a TSL file is generated based on the selections made in the user interface. The TSL file is saved together with the assignment in the database. After the automated assessment is configured and saved, the parser reads in the TSL file and transforms it into a Kotlin tree object consisting of relevant details of the assessment. After the intermediate tree object is created, it is fed to the compiler that transforms it into executable Python code consisting of function calls from the Tiivad library that take care of assessing the uploaded submission.

Once an assignment has been created together with its automated assessment, it is ready to be tested and used by the students. Students can upload their solution to the selected assignment for automated assessment. The student's submitted code is sent together with the generated Tiivad assessment code into an isolated Docker container. Such design ensures that assessment takes place in a secure and controlled manner and prevents potential security threats. The Tiivad back-end assessment code is executed in the Docker container, and it performs the assessment on the submitted solution and returns the results. The results together with feedback are processed, saved into the database and also returned to the user interface where they will be available for both students and teachers to view.

This design ensures a secure and reliable automated assessment and provides nearly instant results and feedback for students while significantly reducing the workload of the teaching staff. Having the assessment in isolated Docker containers and the assessment definitions described in TSL improves the maintainability

[20] https://github.com/kspar/easy

```
from tiivad import *

validate_files(["""submission.py"""])

execute_test(
    file_name="""submission.py""",
    standard_input_data=["""User input from keyboard"""],
    input_files=[("""inputFile.txt""", """Content of file\nLine2\nLine3""")],
    standard_output_checks=[
        {
            "check_type": """ANY_OF_THESE""",
            "nothing_else": None,
            "expected_value": ["""down""", """up"""],
            "elements_ordered": False,
            "before_message": """Verify if the program's output contains keyword.""",
            "passed_message": """Program's output contains keyword.""",
            "failed_message": """Program's output doesn't contain keyword.""",
            "data_category": """EQUALS""",
            "ignore_case": False,
        }
    ],
    output_file_checks=[
        {
            "file_name": """outputFile.txt""",
            "check_type": """ALL_OF_THESE""",
            "nothing_else": None,
            "expected_value": [
                """Expected content of generated file""",
                """Line2""",
                """Line3""",
                """Line4""",
            ],
            "elements_ordered": False,
            "before_message": """Verify if the program's output file contains keyword.""",
            "passed_message": """Program's output file contains the keyword.""",
            "failed_message": """Program's output file doesn't contain the keyword.""",
            "data_category": """EQUALS""",
            "ignore_case": False,
        }
    ],
    exception_check={
        "expected_value": False,
        "before_message": """Verify that program doesn't throw an exception.""",
        "passed_message": """Program didn't throw an exception.""",
        "failed_message": """Program threw an exception.""",
    },
    type="""program_execution_test""",
    points_weight=1.0,
    id=1,
    name="""Program execution test.""",
    inputs=None,
    passed_next=None,
    failed_next=None,
    visible_to_user=True,
)

print(Results(None))
```

Figure 19. Actual Tiivad assessment code generated by the compiler based on a previously parsed TSL file.

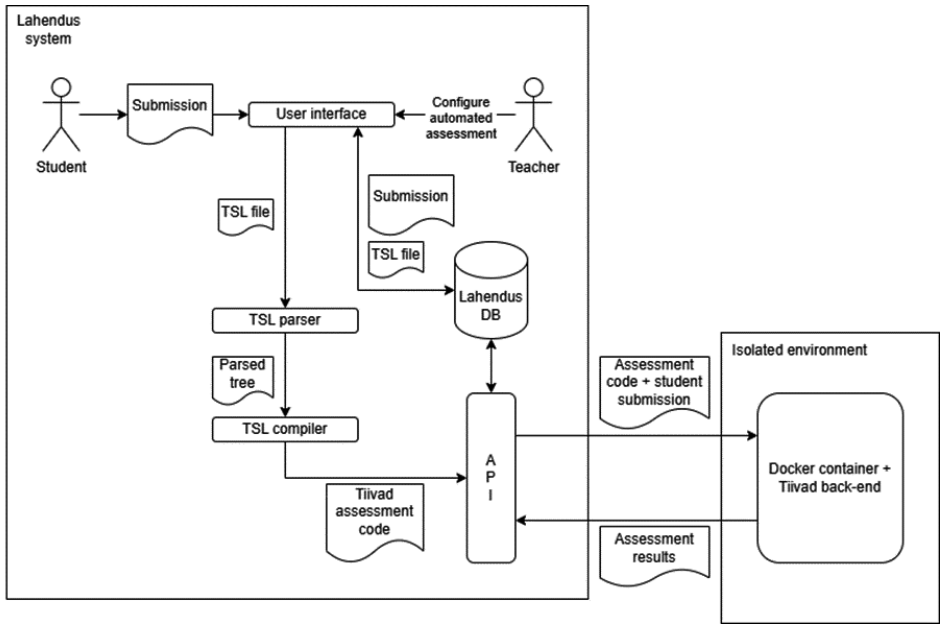

Figure 20. System design scheme showing how different parts of the system communicate with each other.

and scalability of the system. Such a scalable approach can then be utilized in both regular programming courses and also in the large-scale MOOCs.

6.2.6. User interface

The User Interface (UI) is arguably the most critical component of such a system as it serves as the primary interface between the end-user and the back-end services performing the automated assessment. This means that every detail in the assessment configuration within the GUI must be thoroughly considered to avoid incorrectly configured automated assessment resulting in either wrong positives or wrong negatives. That being said, the UI cannot be too comprehensive as it still must be intuitive for the end-users meaning that it should be as simple as possible, but at the same time contain all the necessary details for properly configuring automated assessment. The UI provides capabilities for creating new automated assessment suites consisting of one or more dynamic and static test cases (Figure 21).

Users can select and enable any number of specific tests per assignment, with the option to include the same test multiple times, each with different inputs and expected outputs for better coverage. Each test can have a different custom points weight, making it possible to prioritize some tests over others in terms of points assigned and received. Moreover, different informative messages can be configured, with custom keywords based on test outcome such as failing the test or passing the test, or even for pre-execution (Figure 22 & Figure 23).

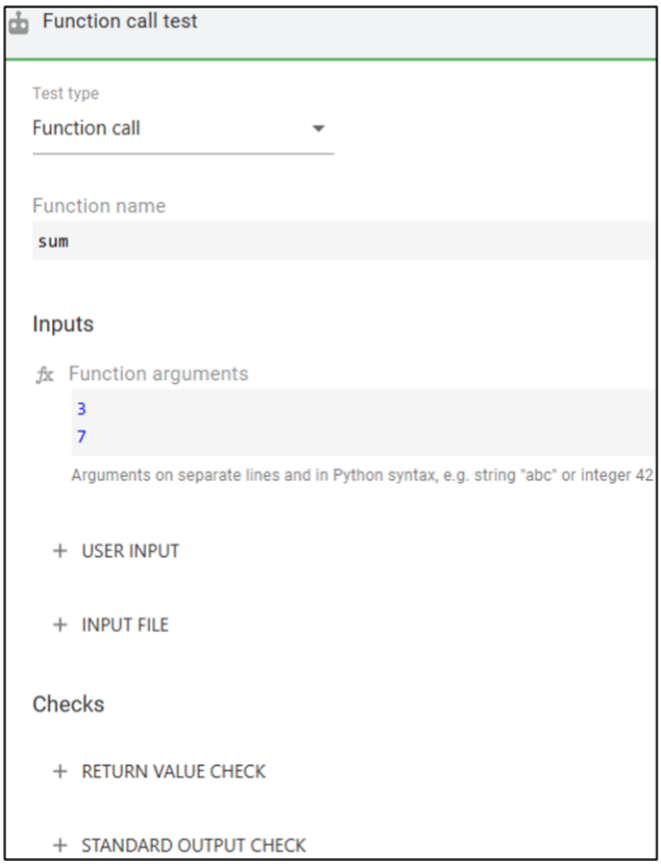

Figure 21. The UI for creation of a test case for validating a function call.

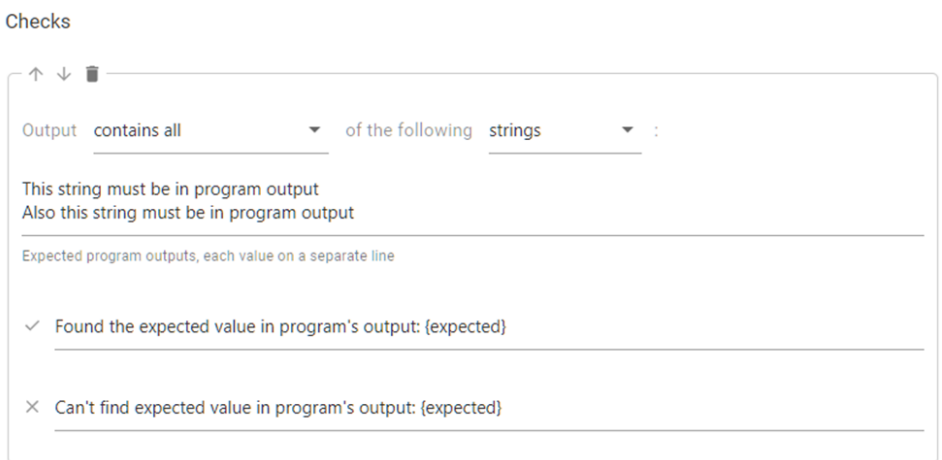

Figure 22. Sample check in program execution test using a custom keyword in response messages.

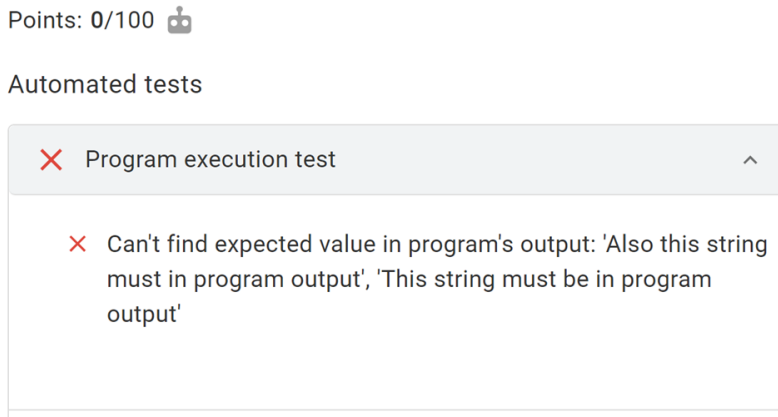

Figure 23. Result of a program execution test in which the keyword of the response message has been replaced with the values provided to the check (Figure 22).

6.3. Validation

The system was thoroughly covered with unit tests and in addition to this, it was validated on real solutions before piloting it on a real course. The idea behind validating on real solutions was to collect a set of different approaches that can be used for solving an exercise and to figure out different edge cases that the unit tests would not cover. In total, around 14400 submissions (both correct and incorrect) were used among 15 different assignments. All the 15 assignments were moved to the new system together with their respective automated assessments that were made compatible with the new system. Such an approach made it possible to validate and compare the results for the same solutions on both systems. A simple Python script was created that took care of the comparison. It ran all the submissions through both the new and old automated assessment systems and built comparative Excel tables out of the results for each assignment. Such an approach found several issues in the old system and more than 50 bugs in the new system that were not discovered by the unit tests.

For example, some specific code structures such as "time.sleep()", "sys.exit()", and "if name == 'main':" would disrupt the normal assessment procedure and cause endless loops and premature termination of assessment. Such issues in a real course would potentially reduce the benefits gained through automated assessment by causing more misunderstandings and false negative grades. Different approaches and solutions to different assignments also caused issues. More specifically, features like importing a package with an alias and creating lists and dictionaries via comprehension were overlooked during the implementation of automated assessment, and any solutions containing these features were not accepted because the assessment library was unable to handle them. Some issues were also caused by the differences in editors, operating systems, and file encodings used by students which had not been accounted for before. Moreover, many

improvements had to be made in TSL, the compiler and the automated assessment code to address gaps in the necessary functionality for testing.

6.4. Evaluation of the system

The new system has been operational since September 2023 and the first results were gathered after three months of use, in December 2023. In this short period of time, it has already been used to create automated assessments for around 115 programming tasks across various introductory programming courses at the University of Tartu. Some of the courses have been Introduction to Programming, Introduction to Programming II, and Computer Programming. There are already plans to expand it to more courses. In total, more than 500 individual checks were created within these first three months of usage.

Creating an automated assessment for a simple task, such as checking whether a program contains a loop, required writing more than 100 lines of custom assessment code in the old system based on Moodle. Moreover, for each of the automated assessments, the assessment library had to be attached manually. In addition, every time there was an improvement or fix in the assessment library, it had to be updated in each and every assignment separately (in case the fix or improvement was relevant). As a result, the latest version of an assessment library is often difficult to locate, after it has spread to multiple branches over time whilst new assessments were created. The process of creating a new automated assessment for any given programming task could take up to two hours. This time could easily double if the person was not properly onboarded or lacked the technical expertise. On the other hand, creating a simple automated assessment in the new system via the user interface can be done in less than 10 clicks and in a couple of minutes. Such a great difference in time and complexity shows the effectiveness and benefits of the new system right away.

Even though the new system for creating and maintaining automated assessments is more user-friendly and saves time, it does not remove all the overhead from the teaching staff. The key questions on what needs to be tested in which programming tasks and on designing the test cases remain the same. That being said, the overhead and complexity of testing is now reduced to selecting options from the user interface instead of writing custom assessment code, which makes the whole process less time consuming and tedious. As mentioned before, the assessment code in the old system was written in Python and the median size of an automated assessment was 4KB and 117 lines of code (excluding common modules and assessment library). Implementation and testing for each task took approximately 1-2 hours on average. In contrast, the creation of similar assessments in TSL via the user interface takes about 5-15 minutes depending on the amount of test cases. What makes the new system even more powerful is the fact that no custom validation or testing of the automated assessment has to be performed since there is little to no room for error in case of making the correct

and valid selection in the user interface. Analysis reveals that the estimated time saved by using the new system was approximately 170 hours, based on a comparison of task creation times between the old system (1 to 2 hours per task) and the new system (5 to 15 minutes per task). Although this calculation provides a reasonable estimate, it is based on user-reported experiences and may not be precise 100%. The actual time savings may vary depending on the complexity of tasks and individual user workflows.

If a user chooses to write TSL manually (instead of clicking on selections via UI) in the Lahendus environment, the system has a feature that validates the correctness of TSL and provides hints in case of malformed TSL, for instance, required fields missing or unknown fields. This ensures that the manually created TSL is syntactically correct, and it can only be saved and attached to the task if it is actually correct. This significantly reduces the time required for testing automated assessment logic and minimizes the need to use any reference materials or look at other automated assessments during implementation. At this moment, the biggest gains can be seen while creating automated assessment for object-oriented programming assignments. In the old system, the automated assessment code written in Python for OOP tasks averaged about 30 KB and 690 lines, because they often consist of multiple classes containing several fields and methods.

Overall, the implementation of TSL for automated assessment offers a stable and efficient solution for solving many of the challenges faced by the teaching staff. TSL is standardized, user-friendly and robust in the sense of error resistance. It can be a valuable tool for improving the creation and maintenance of automated assessments for programming tasks.

7. DISCUSSION

This chapter provides an overview of the discussion of the key findings of this research based on the research questions. Each of the findings is also discussed in more detail within their respective Publications I-IV.

The main motivation for this research was the increasing load on the teaching staff due to growing student enrollments in IT-related courses. Traditional methods of assessment and feedback are effective, but only for small courses with a small number of participants. When the number of participants increases, such manual methods become impractical when applied at scale. The key focus for this thesis was to explore how automation could address these challenges by improving the efficiency, scalability and reliability of the assessment processes while making sure that the quality of assessment remains valuable.

7.1. Teaching staff perspectives on assessment and feedback automation

A crucial aspect of this thesis was the involvement of faculty members through interviews in order to identify the key challenges they faced in assessment and feedback (RQ.1.1). These discussions revealed two recurring pain points. Firstly, there is a large amount of time spent manually grading graphical programming assignments due to the lack of automated tools capable of evaluating visual outputs. The second point is the complexity of creating and maintaining automated assessments using existing tools. The existing tools often required technical knowledge or a lot of manual configuration, which made them impractical for non-technical teaching staff. These insights were directly used as a basis for the other two technical contributions of this research. First is the automated assessment system for graphical programming tasks that utilizes image recognition. Second, the TSL framework simplifies the creation and maintenance of automated assessments. This shows how research can be based on real-world teaching staff needs rather than purely theoretical considerations.

Furthermore, faculty members provided valuable insights into what types of feedback are most effective in programming education (RQ.1.2). Across the interviews, it was clear that students benefit most from error-specific feedback (knowledge about mistakes, KM) and step-by-step debugging guidance (knowledge about how to proceed, KH). These findings align with research stating that effective feedback should be actionable and informative rather than simply confirming correctness (Narciss, 2008; Keuning et al., 2018). Interestingly, while many existing automated assessment tools are capable of binary correctness feedback (knowledge of results, KR), faculty members expressed concerns that such simplistic feedback does not support meaningful learning. This aligns with previous studies (Ala-Mutka et al., 2004; Pieterse, 2013), highlighting that students require deeper feedback explanations to improve their coding skills. Faculty mem-

bers also recognized the value of hints and task-specific guidance (knowledge about task constraints, KTC). Still, they stressed that these should be used cautiously to prevent relying solely on automated hints.

7.2. Automating the assessment of programming tasks with graphical output

One of the most urgent issues raised by faculty members during interviews was the lack of automated assessment for programming tasks with graphical output. In text-based programming tasks, correctness can be determined through structured test cases (Paiva et al., 2022). Still, graphical assignments require visual inspection to determine if the generated graphical output corresponds to the expectations. This process makes the graphical assignments highly labor-intensive. Given these challenges, this study examined how image recognition technology could be leveraged to automatically assess programming tasks with graphical output (RQ.2.1).

Prior research on automated assessment has mostly focused on text-based evaluation, with limited exploration of graphical task automation (English, 2004; Thornton et al., 2008). Since manual grading of graphical outputs is highly time-consuming and existing solutions are inadequate, a new system was developed to automate grading by detecting objects in student-generated images using image recognition. Previous research on graphical assessments (Mertz et al., 2008; Tisha et al., 2023) introduced automated assessment for graphical tasks, but existing solutions did not fully automate the process of generating graphical output from student code and assessing it in an educational setting. This study demonstrates that such an automated assessment system can be implemented, bridging this gap and enabling scalable assessment of graphical programming tasks.

It was equally important to evaluate the system's performance and reliability (RQ.2.2). The system had 4.6% false negatives and 0.5% false positives, mainly due to object detection errors. The system received a 4.4/5 score from its users and reduced manual assessment efforts by 28 hours for instructors, which shows both effectiveness and potential for further improvements. However, challenges remain, particularly in ensuring that the system is robust enough to handle diverse graphical representations while maintaining fairness and accuracy. The effectiveness of the system depends on the quality of image recognition models, which may struggle with complex or unconventional student solutions. These findings suggest that while automated graphical assessment represents a significant step forward, further refinement is needed to ensure that such systems can effectively balance automation with human oversight.

7.3. Simplifying the creation and maintenance of automated assessment with TSL

Another major challenge highlighted by faculty members was the difficulty of setting up automated assessments using existing tools. Many instructors expressed frustration with the technical complexity required to create and maintain automated assessments (RQ.3.1). This aligns with prior research on the challenges of designing automated assessment systems, including considerations for teaching practices, assessment policies, and platform security (Edwards & Pérez-Quinones, 2008; Skalka et al., 2019). These challenges can discourage teaching staff from adopting these tools despite their potential benefits. In direct response to these concerns, this study created the Test Specific Language (TSL) — a standardized, language-agnostic framework designed to simplify the creation and maintenance of automated assessments. TSL provides a structured but intuitive approach that allows educators to define automated assessments without requiring deep programming expertise. By lowering technical barriers, TSL makes automated assessment more accessible, especially for instructors without a programming background. The study's findings confirm that a structured approach to defining assessments can significantly reduce the load on educators, enabling them to focus more on pedagogy rather than on technical implementation (RQ.3.2). By making the automated assessment creation and maintenance more user-friendly, TSL represents a step toward making it more practical, sustainable, and scalable in IT education.

One of the challenges in educational technology is to ensure the long-term sustainability of software tools. Many automated assessment systems are developed as research projects but fail due to limited maintenance resources, evolving technologies, and lack of standardization. Haaranen et al. (2023) highlight that many computing education tools suffer from software decay and often become obsolete unless they are actively maintained and aligned with long-term institutional goals. However, the system developed in this research has already been successfully integrated into the existing Lahendus system, which is used in hundreds of schools in Estonia. Lahendus has a dedicated development team responsible for ongoing maintenance, updates, and long-term support to ensure that the system remains functional and up to date. Furthermore, at the university level, multiple students are actively working on research projects (bachelor and master's theses) that contribute to the continued improvement of TSL and related tools. This approach mitigates the risk of abandonment, ensuring institutional support and community-driven development.

In conclusion, this study highlights the importance of designing scalable and effective automated assessment systems that have been designed and built from the end-users' perspective. In the future, assessment tools utilizing AI and machine learning mechanisms for providing feedback can provide personalized, adaptive feedback in which the responses evolve based on student progress (Gupta et

al., 2019; Bhatia et al., 2018). The findings of this research lay a foundation for future advancements in automated assessment, with implications for both academia and the broader educational technology landscape.

8. CONCLUSION AND IMPLICATIONS

In this chapter, the implications and limitations of this study are presented together with suggestions for future research. This thesis used a qualitative approach in the form of mini-group interviews and inductive content analysis in order to capture the perspectives of the teaching staff regarding automated assessment and feedback. A system capable of automatically assessing programming tasks with graphical output was developed in this research and also validated by gathering the insights and results from the participants. Another system consisting of TSL, its parser and compiler, assessment library Tiivad, and a user interface that orchestrates the actions, was implemented within this research in order to standardize and simplify the creation and maintenance of automated assessments. The results of the thesis provide valuable insights for educators in educational institutions to simplify and support the teaching and learning processes such as assessment and feedbacking. Additionally, the developed systems and insights gathered from the teaching staff can be used as a framework for future research and development.

8.1. Theoretical implications

The theoretical implications of this research are outlined below.

Understanding teaching staff perspectives on automated assessment systems. The perspectives of teaching staff were captured in regard to automated assessment systems. The results help to get insights into the benefits and challenges related to such systems and technologies. The faculty members highlighted multiple important work processes that could be automated or for which automation could be improved, such as grading programming assignments. They also suggested that it is crucial to find the right balance between automation and human interaction, recommending the use of automation for home assignments where instant human feedback is not possible. It was highlighted that automation is more challenging when it comes to creative assignments and it might be more beneficial to assess assignments incrementally instead of only comparing the result with the expected outcome. Different submission strategies and policies were discussed in regard to forcing students to engage more with the assignment at hand and not rely too much on automation. One of the concerning outcomes was that there is currently no easy way to identify struggling students since there are no overview dashboards or automated notifications indicating who may be struggling.

The role and improvement of automated feedback systems. The opinions of the teaching staff also emphasized the importance of automated feedback systems. It was mentioned that automated feedback must be detailed but then again student-friendly, and it should be able to provide feedback on both the correct and incorrect solutions. The faculty members indicated that timely and appropriate feedback is crucial for students' learning and progress. Automated feedback

should help students understand their mistakes and guide them on how to fix those errors without telling them the exact correct answer to the solution. It was recommended to improve the automated feedback to make it as close to natural language as possible, so that it would not scare students away by being too technical and detailed and would also feel similar to the feedback they receive from the teaching staff. The faculty members also mentioned that both students and teachers could benefit if the erroneous spot in the submitted solution would be highlighted in a different color. In addition, it was suggested that it would be useful and interesting to see the ongoing progress of students in the labs while they are solving the assignments in order to capture the places where they might need help or get stuck.

8.2. Practical implications

The practical implications of this research are outlined below.

Developing automated assessment systems to enhance programming education. A new system capable of automatically assessing programming assignments with graphical output was developed and tested out on real courses. The system is using an image recognition service provider, called Clarifai, to recognize objects in the images generated from the submitted solutions. The image recognition service rates the provided images with a score indicating the probability that the given object is present in the provided image. The probability score of 0.7 (70%) is used as the passing threshold, but this can be configured based on assignment and any specific needs. This system helps to reduce the manual workload of instructors and is an efficient and scalable solution that can be used in different courses.

Enhancing assessment reliability and usability through Test Specific Language (TSL). Test Specific Language (TSL) was developed to standardize and simplify the creation and maintenance of automated assessments for programming tasks. In addition to TSL, the following tools were developed to integrate TSL into an already working solution: A parser that verifies the correctness of the given TSL file, a compiler that generates executable automated assessment scripts for the Tiivad library, and a user interface that simplifies the creation of automated assessments by making it possible to configure the assessments via dropdowns, checkboxes and text fields. The Tiivad back-end framework takes care of the actual assessment and feedback within an isolated environment in Docker. This system simplifies the creation and maintenance of automated assessments and also reduces the manual workload of the teaching staff significantly. Furthermore, it minimizes the risk of (human) errors and improves the consistency among different courses and assignments.

8.3. Limitations

This research has several limitations, which are discussed below.

Scope and generalizability of the study. The interviews conducted in this study were limited to a single specialty chair at a single university. This potentially reduces the generalizability of the findings regarding automated assessment and feedback practices for different educational institutions and environments. Additionally, the study focuses primarily on the perspectives and opinions of the teaching staff and leaves out the perspectives of the students. Since students have also a central place in the teaching and learning processes, their insights and experiences with automation could provide valuable information and ideas. Furthermore, the interviews of this research were all carried out in a short and limited timeframe, which could potentially leave out trends that are relevant in a different semester or time of the year and might also exclude a long-term vision.

Dependence on a single image-recognition service provider. The automated assessment for programming tasks with graphical output is currently relying on a single image-recognition service provider (Clarifai). There are other alternatives available in the market that could potentially provide better accuracy and more functionalities. The technology stack has a hard requirement, meaning that the students have to use the Tkinter library for generating the graphical output, which might not be applicable in different educational settings.

Limited functionality and language support in Test Specific Language (TSL). The current implementation of Test Specific Language (TSL) supports a limited number of tests and checks and only the Python programming language. Having support for only a single programming language could reduce the appeal for using TSL in situations where, for example, the course has already been set up for other technologies. Additionally, TSL has been used and tested within a single educational environment, Lahendus, which limits the scope and generalizability of its applicability and usability.

8.4. Suggestions & future research

The following section outlines suggestions for future research and potential improvements.

Expanding the study to multiple institutions and student perspectives. Expanding the scope of the study to include multiple institutions and faculties, and also their students, would provide a wider range of insights into the field of automated assessment systems. Also, having a broader perspective would help to capture common areas of focus and also common challenges in different educational settings. Having such a common ground would also help to solve those challenges in a more uniformly applicable manner instead of tackling the issues individually in each institution.

Investigating the impact of automation on teaching and learning. Future

research could also study and measure the impact of (improved) automation (including automated assessment and feedback). It would be interesting to know how automation affects the teaching methodology from the teachers' point of view and how it affects the learning processes from the students' point of view. Such research could help us find out how much and what kind of influence do these automated systems have on teaching strategies, engagement, and learning outcomes. With more information on those aspects, it would be easier to improve the overall user experience of the systems and also simplify and ease the adoption of such systems and technologies.

Enhancing automation for programming tasks with graphical output. There are multiple ways how the automation for programming tasks with graphical output could be researched further and improved. For example, more image recognition service providers could be analyzed and potentially the previous years' submissions could be used as input data and training set for the service providers in order to improve the accuracy. Also, there might be new technologies and functionalities available that could be used in addition to merely recognizing objects in images, for example, detecting different colors in an image and perhaps even telling the user what was detected in the image instead of only trying to identify objects. Furthermore, research could be done on the ways the feedback provided to the student could be improved above and beyond of only indicating binarily whether the result was correct or not.

Expanding Test Specific Language (TSL) for broader applicability. Future research on TSL could focus on improving its adaptability and applicability in different educational settings. For example, since it currently supports only the Python programming language, it could be extended to add support for more programming languages, which would make TSL more attractive for different courses and curricula. Furthermore, expanding the list of available tests and checks would help to cover a wider range of potentially more complex topics, which would reduce the load of manual assessment for those topics and make the assessment more consistent between different courses and assignments.

Exploring the potential of AI in automated assessment and feedback. The number of use cases and capabilities that Artificial Intelligence (AI) is capable of solving or assisting in is constantly growing. Different AI technologies and applications could potentially be used in educational contexts to provide more sophisticated and student-specific feedback. AI could analyze different patterns in students' submissions and provide more individual suggestions for personalized learning needs. Future research should investigate and explore the different capabilities and potential of AI in terms of automated assessment and feedback in order to provide feedback customized for the particular student and the assignment.

BIBLIOGRAPHY

Abiteboul, S., Agrawal, R., Bernstein, P., Carey, M., Ceri, S., Croft, B., DeWitt, D., Franklin, M., Molina, H. G., Gawlick, D., Gray, J., Haas, L., Halevy, A., Hellerstein, J., Ioannidis, Y., Kersten, M., Pazzani, M., Lesk, M., Maier, D., … Zdonik, S. (2005). The Lowell database research self-assessment. *Communications of the ACM, 48*(5), 111–118. https://doi.org/10.1145/1060710.1060718

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., & Gulwani, S. (2018). Compilation error repair: For the student programs, from the student programs. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 78–87. https://doi.org/10.1145/3183377.3183383

Ahtiainen, A., Surakka, S., & Rahikainen, M. (2006). Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, 141–142. https://doi.org/10.1145/1315803.1315831

Aiken, A. (2002). *MOSS, A System for Detecting Software Plagiarism.* Stanford University. `https://theory.stanford.edu/~aiken/moss/`

Ala-Mutka, K. M. (2005). A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education, 15*(2), 83–102. https://doi.org/10.1080/08993400500150747

Ala-Mutka, K., Uimonen, T., & Järvinen, H.-M. (2004). Supporting Students in C++ Programming Courses with Automatic Program Style Assessment. *Journal of Information Technology Education: Research, 3*(1), 245–262.

Allowatt, A., & Edwards, S. H. (2005). IDE Support for test-driven development and automated grading in both Java and C++. *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, 100–104. https://doi.org/10.1145/1117696.1117717

Alstes, A., & Lindqvist, J. (2007). VERKKOKE: Learning routing and network programming online. *ACM SIGCSE Bulletin, 39*(3), 91–95. https://doi.org/10.1145/1269900.1268813

Amelung, M., Forbrig, P., & Rösner, D. (2008). Towards generic and flexible web services for e-assessment. *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, 219–224. https://doi.org/10.1145/1384271.1384330

Amelung, M., Piotrowski, M., & Rösner, D. (2006). EduComponents: Experiences in e-assessment in computer science education. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 88–92. https://doi.org/10.1145/1140124.1140150

Anderson, G. J. (1990). *Fundamentals of educational research*. London; New York: Falmer.

Annor, P. S., Kayang, E., Boateng, S., & Boateng, G. (2022). AutoGrad: Automated Grading Software for Mobile Game Assignments in SuaCode Courses. *Proceedings of the 10th Computer Science Education Research Conference*, 79–85. https://doi.org/10.1145/3507923.3507954

Antoniol, G., Casazza, G., Penta, M. D., & Fiutem, R. (2001). Object-oriented design patterns recovery. *Journal of Systems and Software, 59*(2), 181–196. https://doi.org/10.1016/S0164-1212(01)00061-9

Anzai, K., & Watanobe, Y. (2019). Algorithm to Determine Extended Edit Distance between Program Codes. *2019 IEEE 13th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC)*, 180–186. https://doi.org/10.1109/MCSoC.2019.00033

Azaiz, I., Deckarm, O., & Strickroth, S. (2023). AI-enhanced auto-correction of programming exercises: How effective is GPT-3.5? *International Journal of Engineering Pedagogy (iJEP), 13*(8), 67–83. https://doi.org/10.3991/ijep.v13i8.45621

Baharum, A., Moung, E., Ismail, I., Ismail, R., Majalin, M., & Mat Noor, N. A. (2020). A Preliminary Study of Difficulties in Learning Java Programming for Secondary School. *International Journal of Advanced Trends in Computer Science and Engineering, 9*, 302–306. https://doi.org/10.30534/ijatcse/2020/4591.42020

Barr, V., & Guzdial, M. (2016). Introducing CS to newcomers, and JES as a teaching tool. *Communications of the ACM, 59*(11), 10–11. https://doi.org/10.1145/2994590

Barra, E., López-Pernas, S., Alonso, Á., Sánchez-Rada, J. F., Gordillo, A., & Quemada, J. (2020). Automated Assessment in Programming Courses: A Case Study during the COVID-19 Era. *Sustainability, 12*(18), 7451. https://doi.org/10.3390/su12187451

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023). Programming is hard – Or at least it used to be: Educational

opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)* (pp. 500–506). Association for Computing Machinery. https://doi.org/10.1145/3545945.3569759

Benford, S. D., Burke, E. K., Foxley, E., & Higgins, C. A. (1995). The Ceilidh system for the automatic grading of students on programming courses. *Proceedings of the 33rd Annual on Southeast Regional Conference,* 176–182. https://doi.org/10.1145/1122018.1122050

Berry, R. E. (1966). Grader Programs. *The Computer Journal, 9*(3), 252–256. https://doi.org/10.1093/comjnl/9.3.252

Bettini, L., Crescenzi, P., Innocenti, G., & Loreti, M. (2004). *An environment for self-assessing Java programming skills in first programming courses* (p. 165). https://doi.org/10.1109/ICALT.2004.1357395

Bhatia, S., Kohli, P., & Singh, R. (2018). Neuro-symbolic program corrector for introductory programming assignments. *Proceedings of the 40th International Conference on Software Engineering*, 60–70. https://doi.org/10.1145/3180155.3180219

Böszörményi, L. (1998). Why Java is not my favorite first-course language. *Software - Concepts & Tools, 19*(3), 141–145. https://doi.org/10.1007/s003780050017

Brusilovsky, P., & Sosnovsky, S. (2005). Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK. *Journal on Educational Resources in Computing, 5*(3), 6-es. https://doi.org/10.1145/1163405.1163411

Caulo, M., Francese, R., Scanniello, G., & Tortora, G. (2021). Relationships between Personality Traits and Productivity in a Multi-platform Development Context. *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, 70–79. https://doi.org/10.1145/3463274.3463327

Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., & Im, E. G. (2013). Software plagiarism detection: A graph-based approach. *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, 1577–1580. https://doi.org/10.1145/2505515.2507848

Chang, C. I., Choi, W. C., & Choi, I. C. (2025). A systematic literature review of the opportunities and advantages for AIGC (OpenAI ChatGPT, Copilot, Codex) in programming course. In *Proceedings of the 2024 7th International Conference on Big Data and Education (ICBDE '24)* (pp. 29–35). Association for Computing

Machinery. https://doi.org/10.1145/3704289.3704301

Charitsis, C., Piech, C., & Mitchell, J. C. (2022). Simplifying Automated Assessment in CS1. *Proceedings of the 2021 4th International Conference on Education Technology Management*, 226–231. https://doi.org/10.1145/3510309.3510345

Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education, 41*(2), 121–131. https://doi.org/10.1016/S0360-1315(03)00030-7

Chelf, B., & Ebert, C. (2009). Ensuring the Integrity of Embedded Software with Static Code Analysis. *IEEE Software, 26*(3), 96–99. IEEE Software. https://doi.org/10.1109/MS.2009.65

Chen, C., Kang, J. M., Sonnert, G., & Sadler, P. M. (2021). High School Calculus and Computer Science Course Taking as Predictors of Success in Introductory College Computer Science. *ACM Transactions on Computing Education, 21*(1), 6:1-6:21. https://doi.org/10.1145/3433169

Chen, H.-M., Chen, W.-H., Hsueh, N.-L., Lee, C.-C., & Li, C.-H. (2017). ProgEdu — An automatic assessment platform for programming courses. *2017 International Conference on Applied System Innovation (ICASI)*, 173–176. https://doi.org/10.1109/ICASI.2017.7988376

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pondé, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating large language models trained on code. *ArXiv, abs/2107.03374*. https://arxiv.org/abs/2107.03374

Chen, M.-Y., Wei, J.-D., Huang, J.-H., & Lee, D. T. (2006). Design and applications of an algorithm benchmark system in a computational problem solving environment. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 123–127. https://doi.org/10.1145/1140124.1140159

Chen, P. M. (2004). An automated feedback system for computer organization projects. *IEEE Trans. on Educ., 47*(2), 232–240. https://doi.org/10.1109/TE.2004.825220

Chen, R., Hong, L., Chunyan Lü, C., & Deng, W. (2010). Author Identification of Software Source Code with Program Dependence Graphs. *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, 281–286.

https://doi.org/10.1109/COMPSACW.2010.56

Chickering, A., & Gamson, Z. (2006). Seven Principles of Good Practice in Undergraduate Education. *New Directions for Teaching and Learning, 1991*, 63–69. https://doi.org/10.1002/tl.37219914708

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493. https://doi.org/10.1109/32.295895

Chisăliţă-Creţu, C., Bota, F., & Pop, A.-D. (2021). Software Testing Education Experiences Using Collaborative Platforms. In V. L. Uskov, R. J. Howlett, & L. C. Jain (Eds.), *Smart Education and e-Learning 2021* (pp. 149–159). Springer. https://doi.org/10.1007/978-981-16-2834-4_13

Close, R., Kopec, D., & Aman, J. (2000). CS1: Perspectives on programming languages and the breadth-first approach. *J. Comput. Sci. Coll., 15*(5), 228–234.

Coffman, J., & Weaver, A. C. (2010). Electronic commerce virtual laboratory. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 92–96. https://doi.org/10.1145/1734263.1734295

Cosma, G., & Joy, M. (2012). An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *IEEE Transactions on Computers, 61*(3), 379–394. IEEE Transactions on Computers. https://doi.org/10.1109/TC.2011.223

Danutama, K., & Liem, I. (2013). Scalable Autograder and LMS Integration. *Procedia Technology, 11*, 388–395. https://doi.org/10.1016/j.protcy.2013.12.207

de Raadt, M., Dekeyser, S., & Lee, T. Y. (2006). Do students SQLify? Improving learning outcomes with peer review and enhanced computer assisted assessment of querying skills. *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, 101–108. https://doi.org/10.1145/1315803.1315821

Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 273–278. https://doi.org/10.1145/2591708.2591748

Denny, P., Prather, J., Becker, B. A., Finnie-Ansley, J., Hellas, A., Leinonen, J., Luxton-Reilly, A., Reeves, B. N., Santos, E. A., & Sarsa, S. (2024). Computing

education in the era of generative AI. *Communications of the ACM, 67*(2), 56–67. https://doi.org/10.1145/3624720

Diana, N., Eagle, M., Stamper, J., Grover, S., Bienkowski, M., & Basu, S. (2017). An instructor dashboard for real-time analytics in interactive programming assignments. *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, 272–279. https://doi.org/10.1145/3027385.3027441

Dong, J., Sun, Y., & Zhao, Y. (2008). Design pattern detection by template matching. *Proceedings of the 2008 ACM Symposium on Applied Computing*, 765–769. https://doi.org/10.1145/1363686.1363864

Dong, T., & Khandwala, K. (2019). The Impact of 'Cosmetic' Changes on the Usability of Error Messages. *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, 1–6. https://doi.org/10.1145/3290607.3312978

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing, 5*(3), 4-es. https://doi.org/10.1145/1163405.1163409

Dromey, R. G., & Ryan, K. (1993). PASS-C: Program analysis and style system user manual. *Software Quality Institute, Griffith University.*

Đurić, Z., & Gašević, D. (2013). A Source Code Similarity System for Plagiarism Detection. *The Computer Journal, 56*(1), 70–86. https://doi.org/10.1093/comjnl/bxs018

Edwards, S. H. (2003). Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing, 3*(3), 1-es. https://doi.org/10.1145/1029994.1029995

Edwards, S. H., & Perez-Quinones, M. A. (2008). Web-CAT: Automatically grading programming assignments. *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, 328. https://doi.org/10.1145/1384271.1384371

Ellsworth, C. C., Fenwick, J. B., & Kurtz, B. L. (2004). The Quiver system. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 205–209. https://doi.org/10.1145/971300.971374

Elo, S., & Kyngäs, H. (2008). The qualitative content analysis process. *Journal of Advanced Nursing, 62*(1), 107–115.

https://doi.org/10.1111/j.1365-2648.2007.04569.x

English, J. (2004). Automated assessment of GUI programs using JEWL. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 137–141. https://doi.org/10.1145/1007996.1008033

Enstrom, E., Kreitz, G., Niemela, F., Soderman, P., & Kann, V. (2011). Five years with kattis—Using an automated assessment system in teaching. *Proceedings of the 2011 Frontiers in Education Conference*, T3J-1-1-T3J-6. https://doi.org/10.1109/FIE.2011.6142931

Eurostat. (2024). *ICT specialists in employment*. European Commission. Retrieved from `https://ec.europa.eu/eurostat/statistics-explained/index.php?title=ICT_specialists_in_employment`

Falkner, N., Vivian, R., Falkner, K., Ajanovski, V. V., Liebe, C., Morrison, A., & Parker, M. (2020). Meaningful Assessment at Scale: Helping Instructors to Assess Online Learning. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, 512–513. https://doi.org/10.1145/3341525.3394993

Fenton, N. E., & Neil, M. (1999). Software metrics: Successes, failures and new directions. *Journal of Systems and Software, 47*(2), 149–157. https://doi.org/10.1016/S0164-1212(99)00035-7

Flores, E., Barrón-Cedeño, A., Moreno, L., & Rosso, P. (2015). Cross-Language Source Code Re-Use Detection Using Latent Semantic Analysis. *JUCS - Journal of Universal Computer Science, 21*(13), Article 13. https://doi.org/10.3217/jucs-021-13-1708

Forsythe, G. E., & Wirth, N. (1965). Automatic grading programs. *Communications of the ACM, 8*(5), 275–278. https://doi.org/10.1145/364914.364937

Fu, D., Xu, Y., Yu, H., & Yang, B. (2017). WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection. *Scientific Programming, 2017*(1), 7809047. https://doi.org/10.1155/2017/7809047

Fu, X., Peltsverger, B., Qian, K., Tao, L., & Liu, J. (2008). APOGEE: Automated project grading and instant feedback system for web based computing. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, 77–81. https://doi.org/10.1145/1352135.1352163

Fu, X., Shimada, A., Ogata, H., Taniguchi, Y., & Suehiro, D. (2017). Real-

time learning analytics for C programming language courses. *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, 280–288. https://doi.org/10.1145/3027385.3027407

Fulcini, T., Coppola, R., Ardito, L., & Torchiano, M. (2023). A Review on Tools, Mechanics, Benefits, and Challenges of Gamified Software Testing. *ACM Comput. Surv., 55*(14s), 310:1-310:37. https://doi.org/10.1145/3582273

Funabiki, N., Matsushima, Y., Nakanishi, T., Watanabe, K., & Amano, N. (2013, March 1). *A Java programming Learning Assistant System using test-driven development method.* https://api.semanticscholar.org/CorpusID:17943489

Gisev, N., Bell, J. S., & Chen, T. F. (2013). Interrater agreement and interrater reliability: Key concepts, approaches, and applications. *Research in Social and Administrative Pharmacy, 9*(3), 330–338. https://doi.org/10.1016/j.sapharm.2012.04.004

Gotel, O., Scharff, C., & Wildenberg, A. (2007). Extending and contributing to an open source web-based system for the assessment of programming problems. *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, 3–12. https://doi.org/10.1145/1294325.1294327

Graßl, I., Geldreich, K., & Fraser, G. (2021). Data-driven Analysis of Gender Differences and Similarities in Scratch Programs. *Proceedings of the 16th Workshop in Primary and Secondary Computing Education*, 1–10. https://doi.org/10.1145/3481312.3481345

Gray, G. R., & Higgins, C. A. (2006). An introspective approach to marking graphical user interfaces. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 43–47. https://doi.org/10.1145/1140124.1140139

Guerreiro, P., & Georgouli, K. (2006). Combating anonymousness in populous CS1 and CS2 courses. *ACM SIGCSE Bulletin, 38*(3), 8–12. https://doi.org/10.1145/1140123.1140130

Gupta, R., Kanade, A., & Shevade, S. (2019). Neural attribution for semantic bug-localization in student programs. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems* (pp. 11884–11894). Curran Associates Inc.

Gutiérrez, E., Trenas, M. A., Ramos, J., Corbera, F., & Romero, S. (2010). A new Moodle module supporting automatic verification of VHDL-based assign-

ments. *Computers & Education, 54*(2), 562–577. https://doi.org/10.1016/j.compedu.2009.09.006

Haaranen, L., Ahrenberg, L., & Hellas, A. (2024). Decades of Striving for Pedagogical and Technological Alignment. *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23)*, Article 23, 1–8. Association for Computing Machinery. https://doi.org/10.1145/3631802.3631809

Halstead, M. H. (1977). *Elements of Software Science*. Elsevier.

Hansen, H., & Ruuska, M. (2003). Assessing Time-Efficiency in a Course on Data Structures and Algorithms. *Proceedings of the Third Finnish / Baltic Sea Conference on Computer Science Education, Oct. 3-5, 2003, Koli, Finland*, 93–100. University of Helsinki, Department of Computer Science.

Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do: Suggesting solutions to error messages. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1019–1028. https://doi.org/10.1145/1753326.1753478

Head, A., Glassman, E., Soares, G., Suzuki, R., Figueredo, L., D'Antoni, L., & Hartmann, B. (2017). Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, 89–98. https://doi.org/10.1145/3051457.3051467

Helmick, M. T. (2007). Interface-based programming assignments and automatic grading of java programs. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 63–67. https://doi.org/10.1145/1268784.1268805

Higgins, C., Hegazy, T., Symeonidis, P., & Tsintsifas, A. (2003). The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies, 8*(3), 287–304. https://doi.org/10.1023/A:1026364126982

Higgins, C., Symeonidis, P., & Tsintsifas, A. (2002). Diagram-based CBA using DATsys and CourseMaster. *International Conference on Computers in Education, 2002. Proceedings.*, 167–172 vol.1. https://doi.org/10.1109/CIE.2002.1185893

Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM, 3*(10), 528–529. https://doi.org/10.1145/367415.367422

Hovemeyer, D., Hellas, A., Petersen, A., & Spacco, J. (2016). Control-Flow-Only

Abstract Syntax Trees for Analyzing Students' Programming Progress. *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 63–72. https://doi.org/10.1145/2960310.2960326

Howles, T. (2003). Fostering the growth of a software quality culture. *ACM SIGCSE Bulletin, 35*(2), 45–47. https://doi.org/10.1145/782941.782978

Huang, Q., Song, X., & Fang, G. (2020). Code Plagiarism Detection Method Based on Code Similarity and Student Behavior Characteristics. *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, 167–172. https://doi.org/10.1109/ICAICA50127.2020.9182389

Hwang, W.-Y., Wang, C.-Y., Hwang, G.-J., Huang, Y.-M., & Huang, S. (2008). A web-based programming learning environment to support cognitive development. *Interacting with Computers, 20*(6), 524–534. https://doi.org/10.1016/j.intcom.2008.07.002

Ihantola, P. (2011). *Automated Assessment of Programming Assignments: Visual Feedback, Assignment Mobility, and Assessment of Students' Testing Skills*. PhD thesis, Aalto University, Finland.

Inoue, U., & Wada, S. (2012). Detecting plagiarisms in elementary programming courses. *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*, 2308–2312. https://doi.org/10.1109/FSKD.2012.6234186

Isaacson, P. C., & Scott, T. A. (1989). Automating the execution of student programs. *ACM SIGCSE Bulletin, 21*(2), 15–22. https://doi.org/10.1145/65738.65741

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

Janhunen, T., Jussila, T., Järvisalo, M., & Oikarinen, E. (2004). Teaching Smullyan's Analytic Tableaux in a Scalable Learning Environment. In *Kolin Kolistelut/Koli Calling: Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education* (pp. 85-94).

Jeuring, J., Keuning, H., Marwan, S., Bouvier, D., Izu, C., Kiesler, N., Lehtinen, T., Lohr, D., Peterson, A., & Sarsa, S. (2022). Towards giving timely formative feedback and hints to novice programmers. *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '22)*, 95–115. https://doi.org/10.1145/3571785.3574124

Jimenez-Gonzalez, D., Alvarez, C., Lopez, D., Parcerisa, J.-M., Alonso, J., Perez, C., Tous, R., Barlet, P., Fernandez, M., & Tubella, J. (2008). Work in progress-improving feedback using an automatic assessment tool. *2008 38th Annual Frontiers in Education Conference*, S3B-9-T1A-10. https://doi.org/10.1109/FIE.2008.4720591

Joy, M., Griffiths, N., & Boyatt, R. (2005). The boss online submission and assessment system. *Journal on Educational Resources in Computing, 5*(3), 2-es. https://doi.org/10.1145/1163405.1163407

Joy, M., & Luck, M. (1998). Effective electronic marking for on-line assessment. *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education*, 134–138. https://doi.org/10.1145/282991.283096

Karvandi, M. S., Gholamrezaei, M., Khalaj Monfared, S., Meghdadizanjani, S., Abbassi, B., Amini, A., Mortazavi, R., Gorgin, S., Rahmati, D., & Schwarz, M. (2022). HyperDbg: Reinventing Hardware-Assisted Debugging. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 1709–1723. https://doi.org/10.1145/3548606.3560649

Kay, D. G., Scott, T., Isaacson, P., & Reek, K. A. (1994). Automated grading assistance for student programs. *ACM SIGCSE Bulletin, 26*(1), 381–382. https://doi.org/10.1145/191033.191184

Ke, H., Zhang, G., & Yan, H. (2009). Automatic Grading System on SQL Programming. *2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, 537–540. https://doi.org/10.1109/EmbeddedCom-ScalCom.2009.105

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv., 37*(2), 83–137. https://doi.org/10.1145/1089733.1089734

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education, 19*(1), Article 3. https://doi.org/10.1145/3231711

Kleiner, C., Tebbe, C., & Heine, F. (2013). Automated grading and tutoring of SQL statements to improve student learning. *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, 161–168. https://doi.org/10.1145/2526968.2526986

Koren, M. (2024). Graphical user interfaces as a method to encourage beginners in learning programming. *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, 1439–1444. https://doi.org/10.1109/MIPRO60963.2024.10569418

Kristiansen, N. G., Nicolajsen, S. M., & Brabrand, C. (2024). Feedback on Student Programming Assignments: Teaching Assistants vs Automated Assessment Tool. *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 1–10. https://doi.org/10.1145/3631802.3631804

Krueger, R. A. (1994). *Focus Groups: A Practical Guide for Applied Research, Second Edition* (2nd edition). SAGE Publications, Inc.

Kwak, M., Jenkins, J., & Kim, J. (2023). Adaptive programming language learning system based on generative AI. *Issues in Information Systems, 24*(3), 222–231. https://doi.org/10.48009/3_iis_2023_119

Lazar, T., Možina, M., & Bratko, I. (2017). Automatic Extraction of AST Patterns for Debugging Student Programs. In E. André, R. Baker, X. Hu, Ma. M. T. Rodrigo, & B. du Boulay (Eds.), *Artificial Intelligence in Education* (pp. 162–174). Springer International Publishing. https://doi.org/10.1007/978-3-319-61425-0_14

Leal, J. P., & Silva, F. (2003). Mooshak: A Web-based multi-site programming contest system. *Software: Practice and Experience, 33*(6), 567–581. https://doi.org/10.1002/spe.522

Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., & Becker, B. A. (2023). Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)* (pp. 563–569). Association for Computing Machinery. https://doi.org/10.1145/3545945.3569770

Leping, V., Lepp, M., Niitsoo, M., Tõnisson, E., Vene, V., & Villems, A. (2009). Python prevails. *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, 1–5. https://doi.org/10.1145/1731740.1731833

Lepp, M., Luik, P., Palts, T., Papli, K., Suviste, R., Säde, M., Hollo, K., Vaherpuu, V., & Tõnisson, E. (2017). Self- and Automated Assessment in Programming MOOCs. In D. Joosten-ten Brinke & M. Laanpere (Eds.), *Technology Enhanced Assessment* (pp. 72–85). Springer International Publishing. https://doi.org/10.1007/978-3-319-57744-9_7

Lepp, M., Luik, P., Palts, T., Papli, K., Suviste, R., Säde, M., & Tõnisson, E. (2017). MOOC in programming: A success story. *Proceedings of the International Conference on E-Learning*, 138–147.

Lepp, M., Palts, T., Luik, P., Papli, K., Suviste, R., Säde, M., Hollo, K., Vaherpuu, V., & Tõnisson, E. (2018). Troubleshooters for Tasks of Introductory Programming MOOCs. *The International Review of Research in Open and Distributed Learning, 19*(4). https://doi.org/10.19173/irrodl.v19i4.3639

Lingling, M., Xiaojie, Q., Zhihong, Z., Gang, Z., & Ying, X. (2008). An Assessment Tool for Assembly Language Programming. *2008 International Conference on Computer Science and Software Engineering, 5*, 882–884. https://doi.org/10.1109/CSSE.2008.111

Liu, X., Kim, Y., Cheon, J., & Woo, G. (2019). A Partial Grading Method using Pattern Matching for Programming Assignments. *2019 8th International Conference on Innovation, Communication and Engineering (ICICE)*, 157–160. https://doi.org/10.1109/ICICE49024.2019.9117506

Liu, Z., Liu, T., Li, Q., Luo, W., & Lumetta, S. (2021). *End-to-End Automation of Feedback on Student Assembly Programs* (p. 29). https://doi.org/10.1109/ASE51524.2021.9678837

Ljubovic, V., & Nosovic, N. (2012). Software metrics in student projects. *2012 20th Telecommunications Forum (TELFOR)*, 1464–1467. https://doi.org/10.1109/TELFOR.2012.6419495

Luik, P., Lepp, M., Palts, T., Säde, M., Suviste, R., Tonisson, E., & Gaiduk, M. (2018). Completion of Programming MOOC or Dropping out: Are There any Differences in Motivation? *Proceedings of the 17th European Conference on e-Learning ECEL*, 329-337.

Malmi, L., Karavirta, V., Korhonen, A., & Nikander, J. (2005). Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal on Educational Resources in Computing, 5*(3), 7-es. https://doi.org/10.1145/1163405.1163412

Martins, J., Bezerra, C., Uchôa, A., & Garcia, A. (2021). How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective. *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, 54–63. https://doi.org/10.1145/3474624.3474642

Mayring, P. (2000). Qualitative Content Analysis. *Forum Qualitative Sozialforschung Forum: Qualitative Social Research, 1*(2). https://doi.org/10.17169/fqs-1.2.1089

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Merry, B. (2009). Using a Linux Security Module for contest security. *Olympiads in Informatics, 3*, 67–73.

Mertz, A., Slough, W., & Van Cleave, N. (2008). Using the ACM Java libraries in CS 1. *J. Comput. Sci. Coll., 24*(1), 16–26.

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2024). Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Transactions on Computing Education, 24*(1), 1–43. https://doi.org/10.1145/3636515

Mets, U., Viia, A., & Ruusa, L. (2024). *Tulevikuvaade tööjõu- ja oskuste vajadusele: Info- ja kommunikatsioonitehnoloogia. Seirearuanne*. SA Kutsekoda. https://uuringud.oska.kutsekoda.ee/uuringud/ikt-seire

Millsap, C. (2010). Thinking Clearly about Performance: Improving the performance of complex software is difficult, but understanding some fundamental principles can make it easier. *Queue, 8*(9), 10–20. https://doi.org/10.1145/1854039.1854041

Monteiro, R. H. B., de Almeida Souza, M. R., Oliveira, S. R. B., Portela, C. dos S., & de Cristo Lobato, C. E. (2021). The diversity of gamification evaluation in the software engineering education and industry: Trends, comparisons and gaps. *Proceedings of the 43rd International Conference on Software Engineering: Joint Track on Software Engineering Education and Training*, 154–164. https://doi.org/10.1109/ICSE-SEET52601.2021.00025

Morris, D. S. (2003). Automatic grading of student's programming assignments: An interactive process and suite of programs. *33rd Annual Frontiers in Education, 2003. FIE 2003., 3*, S3F-1. https://doi.org/10.1109/FIE.2003.1265998

Nabil, R., Borhan Eldeen, N. E., Mahdy, A., Nader, K., Elbohy, S., & Eliwa, E. (2021). *EvalSeer: An Intelligent Gamified System for Programming Assignments Assessment* (p. 242). https://doi.org/10.1109/MIUCC52538.2021.9447629

Nafa, F., Sreeramareddy, L., Mallapuram, S., & Moulema, P. (2023). Improving

Educational Outcomes: Developing and Assessing Grading System (ProGrader) for Programming Courses. In H. Mori & Y. Asahi (Eds.), *Human Interface and the Management of Information* (pp. 322–342). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-35129-7_24

Narciss, S. (2008). Feedback strategies for interactive learning tasks. In J. M. Spector, M. D. Merrill, J. J. G. van Merriënboer, & M. P. Driscoll (Eds.), *Handbook of research on educational communications and technology (pp. 125–143)*. Mahaw, NJ: Lawrence Erlbaum Associates.

Naur, P. (1964). Automatic grading of students' ALGOL programming. *BIT Numerical Mathematics, 4*(3), 177–188. https://doi.org/10.1007/BF01956028

Nguyen, H., Lim, M., Moore, S., Nyberg, E., Sakr, M., & Stamper, J. (2021). Exploring Metrics for the Analysis of Code Submissions in an Introductory Data Science Course. *LAK21: 11th International Learning Analytics and Knowledge Conference*, 632–638. https://doi.org/10.1145/3448139.3448209

Njoku, A., Amini, M., & Sharafi, Z. (2024). Innovating Coding: Evaluating the Impact of Innovative Thinking in Programming. *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 241–245. https://doi.org/10.1145/3643916.3644397

Nordquist, P. (2007). Providing accurate and timely feedback by automatically grading student programming labs. *J. Comput. Sci. Coll., 23*(2), 16–23.

Novak, J., Krajnc, A., & Žontar, R. (2010). Taxonomy of static code analysis tools. *The 33rd International Convention MIPRO*, 418–422. https://ieeexplore.ieee.org/document/5533417

Novak, M. (2016). Review of source-code plagiarism detection in academia. *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 796–801. https://doi.org/10.1109/MIPRO.2016.7522248

Nunome, A., Hirata, H., Fukuzawa, M., & Shibayama, K. (2010). Development of an e-learning back-end system for code assessment in elementary programming practice. *Proceedings of the 38th Annual ACM SIGUCCS Fall Conference: Navigation and Discovery*, 181–186. https://doi.org/10.1145/1878335.1878381

Oechsle, R., & Barzen, K. (2007). Checking automatically the output of concurrent threads. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 43–47.

https://doi.org/10.1145/1268784.1268799

Oehlhorn, C. E., Laumer, S., & Maier, C. (2019). Sustaining the IT Workforce: A Review of Major Issues in 25 Years and Future Directions. *Proceedings of the 2019 on Computers and People Research Conference*, 20–27. https://doi.org/10.1145/3322385.3322394

Ozturk, S. (2022). Gamification of exploratory testing process. *Proceedings of the 1st International Workshop on Gamification of Software Development, Verification, and Validation*, 14–17. https://doi.org/10.1145/3548771.3561411

Paiva, J., Leal, J., & Figueira, Á. (2022). Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education, 22*. https://doi.org/10.1145/3513140

Palumbo, D. B. (1990). Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research, 60*(1), 65–89. https://doi.org/10.3102/00346543060001065

Panni, F. A. K., & Hoque, A. S. Md. L. (2020). A Model for Automatic Partial Evaluation of SQL Queries. *2020 2nd International Conference on Advanced Information and Communication Technology (ICAICT)*, 240–245. https://doi.org/10.1109/ICAICT51780.2020.9333475

Pape, S., Flake, J., Beckmann, A., & Jürjens, J. (2016). STAGE: A software tool for automatic grading of testing exercises: case study paper. *Proceedings of the 38th International Conference on Software Engineering Companion*, 491–500. https://doi.org/10.1145/2889160.2889203

Parihar, S., Dadachanji, Z., Singh, P. K., Das, R., Karkare, A., & Bhattacharya, A. (2017). Automatic Grading and Feedback using Program Repair for Introductory Programming Courses. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 92–97. https://doi.org/10.1145/3059009.3059026

Paris, M. (2003). Source Code and Text Plagiarism Detection Strategies. *4th Annual Conference of the LTSN Centre for Information and Computer Sciences*, 74–78.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin, 39*(4), 204–223. https://doi.org/10.1145/1345375.1345441

Pettit, R., & Prather, J. (2017). Automated assessment tools: Too many cooks, not enough collaboration. *J. Comput. Sci. Coll., 32*(4), 113–121.

Pettit, R. S., Homer, J., & Gee, R. (2017). Do Enhanced Compiler Error Messages Help Students? Results Inconclusive. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 465–470. https://doi.org/10.1145/3017680.3017768

Peveler, M., Maicus, E., & Cutler, B. (2020). Automated and Manual Grading of Web-Based Assignments. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 1373. https://doi.org/10.1145/3328778.3372682

Phothilimthana, P. M., & Sridhara, S. (2017). High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students? *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 182–187. https://doi.org/10.1145/3059009.3059058

Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., & Guibas, L. (2015). Learning program embeddings to propagate feedback on student code. *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, 1093–1102.

Pieterse, V. (2013). Automated Assessment of Programming Assignments. *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, 45–56.

Prado, B., Bispo, K., & Andrade, R. (2018). *X9: An Obfuscation Resilient Approach for Source Code Plagiarism Detection in Virtual Learning Environments* (p. 524). https://doi.org/10.5220/0006668705170524

Prather, J., Denny, P., Leinonen, J., Becker, B. A., Albluwi, I., Craig, M., Keuning, H., Kiesler, N., Kohn, T., Luxton-Reilly, A., MacNeil, S., Petersen, A., Pettit, R., Reeves, B. N., & Savelka, J. (2023). The robots are here: Navigating the generative AI revolution in computing education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '23)* (pp. 108–159). Association for Computing Machinery. https://doi.org/10.1145/3623762.3633499

Prather, J., Pettit, R., Becker, B. A., Denny, P., Loksa, D., Peters, A., Albrecht, Z., & Masci, K. (2019). First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 531–537.

https://doi.org/10.1145/3287324.3287374

Prather, J., Reeves, B. N., Denny, P., Becker, B. A., Leinonen, J., Luxton-Reilly, A., Powell, G., Finnie-Ansley, J., & Santos, E. A. (2023). "It's weird that it knows what I want": Usability and interactions with Copilot for novice programmers. *ACM Transactions on Computer-Human Interaction, 31*(1), Article 4. https://doi.org/10.1145/3617367

Prather, J., Reeves, B. N., Leinonen, J., MacNeil, S., Randrianasolo, A. S., Becker, B. A., Kimmel, B., Wright, J., & Briggs, B. (2024). The widening gap: The benefits and harms of generative AI for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1 (ICER '24)* (pp. 469–486). Association for Computing Machinery. https://doi.org/10.1145/3632620.3671116

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding Plagiarisms among a Set of Programs with JPlag. *JUCS - Journal of Universal Computer Science, 8*(11), Article 11. https://doi.org/10.3217/jucs-008-11-1016

Pugnali, A., Sullivan, A., & Bers, M. (2017). The impact of user interface on young children's computational thinking. *Journal of Information Technology Education: Innovations in Practice, 16*, 171–193. https://doi.org/10.28945/3768

Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., & Binkley, D. (2011). SCOTCH: Test-to-code traceability using slicing and conceptual coupling. *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, 63–72. https://doi.org/10.1109/ICSM.2011.6080773

Rahman, K., & Nordin, M. J. (2007). A Review on the Static Analysis Approach in the Automated Programming Assessment Systems. *Proceedings of the 2007 National Conference on Programming*, 1–15.

Raihan, N., Siddiq, M. L., Santos, J. C. S., & Zampieri, M. (2025). Large language models in computer science education: A systematic literature review. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSETS 2025)* (pp. 938–944). Association for Computing Machinery. https://doi.org/10.1145/3641554.3701863

Ray, B., Posnett, D., Devanbu, P., & Filkov, V. (2017). A large-scale study of programming languages and code quality in GitHub. *Communications of the ACM, 60*(10), 91–100. https://doi.org/10.1145/3126905

Reek, K. A. (1989). The TRY system -or- how to avoid testing student programs.

Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education, 112–116. https://doi.org/10.1145/65293.71198

Rees, M. J. (1982). Automatic assessment aids for Pascal programs. *SIGPLAN Not., 17*(10), 33–42. https://doi.org/10.1145/948086.948088

Rintala, M. (2002). Tutnew—Työkalu C++:n dynaamisen muistinhallinnan testaamiseen. *Tietojenkäsittelytiede, 17*, 8–22.

Rodríguez-del-Pino, J. (2012). A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. *Proceedings of The 2012 Internacional Conference on e-Learning, e-Business, Entreprise Information Systems, & e-Government*.

Romano, S., Zampetti, F., Baldassarre, M. T., Di Penta, M., & Scanniello, G. (2022). Do Static Analysis Tools Affect Software Quality when Using Test-driven Development? *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 80–91. https://doi.org/10.1145/3544902.3546233

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. *ACM SIGCSE Bulletin, 40*(4), 142–166. https://doi.org/10.1145/1473195.1473239

Roy, P. V., Derval, G., Frantzen, B., Gégo, A., & Reinbold, P. (2015). Automatic grading of programming exercises in a MOOC using the INGInious platform. *Proceedings of the European MOOC Stakeholder Summit 2015*, 86-91.

Ruehr, F., & Orr, G. (2002). Interactive program demonstration as a form of student program assessment. *J. Comput. Sci. Coll., 18*(2), 65–78.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision, 115*(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

Saikkonen, R., Malmi, L., & Korhonen, A. (2001). Fully automatic assessment of programming exercises. *ACM SIGCSE Bulletin, 33*(3), 133–136. https://doi.org/10.1145/507758.377666

Sant, J. A. (2009). 'Mailing it in': Email-centric automated assessment. *ACM*

SIGCSE Bulletin, 41(3), 308–312. https://doi.org/10.1145/1595496.1562971

Schorsch, T. (1995). CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *ACM SIGCSE Bulletin, 27*(1), 168–172. https://doi.org/10.1145/199691.199769

Sharma, S., Agarwal, P., Mor, P., & Karkare, A. (2018). TipsC: Tips and Corrections for programming MOOCs. In C. Penstein Rosé, R. Martínez-Maldonado, H. U. Hoppe, R. Luckin, M. Mavrikis, K. Porayska-Pomsta, B. McLaren, & B. du Boulay (Eds.), *Artificial Intelligence in Education* (pp. 322–326). Springer International Publishing. https://doi.org/10.1007/978-3-319-93846-2_60

Shweta, Bajpai, R. C., & Chaturvedi, H. K. (2015). Evaluation of inter-rater agreement and inter-rater reliability for observational data: An overview of concepts and methods. *Journal of the Indian Academy of Applied Psychology, 41*(3), 20–27.

Silva, C. D. S., Ferreira da Costa, L., Rocha, L. S., & Viana, G. V. R. (2020). KNN Applied to PDG for Source Code Similarity Classification. In R. Cerri & R. C. Prati (Eds.), *Intelligent Systems* (pp. 471–482). Springer International Publishing. https://doi.org/10.1007/978-3-030-61380-8_32

Siochi, A. C., & Hardy, W. R. (2015). WebWolf: Towards a Simple Framework for Automated Assessment of Webpage Assignments in an Introductory Web Programming Class. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 84–89. https://doi.org/10.1145/2676723.2677217

Skalka, J., Drlík, M., & Obonya, J. (2019). Automated Assessment in Learning and Teaching Programming Languages using Virtual Learning Environment. *2019 IEEE Global Engineering Education Conference (EDUCON)*, 689-697. https://doi.org/10.1109/EDUCON.2019.8725127

Souza, D. M., Felizardo, K. R., & Barbosa, E. F. (2016). A Systematic Literature Review of Assessment Tools for Programming Assignments. *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, 147–156. https://doi.org/10.1109/CSEET.2016.48

Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., & Padua-Perez, N. (2006). Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 13–17. https://doi.org/10.1145/1140124.1140131

Srikant, S., & Aggarwal, V. (2014). A system to grade computer programming skills using machine learning. *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1887–1896. https://doi.org/10.1145/2623330.2623377

Steinhöfel, D., & Zeller, A. (2024). Language-Based Software Testing. *Communications of the ACM, 67*(4), 80–84. https://doi.org/10.1145/3631520

Strickroth, S., & Striewe, M. (2022). Building a corpus of task-based grading and feedback systems for learning and teaching programming. *International Journal of Engineering Pedagogy, 12*(5), 26–41. https://doi.org/10.3991/ijep.v12i5.31283

Suleman, H. (2008). Automatic marking with Sakai. *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, 229–236. https://doi.org/10.1145/1456659.1456686

Sun, D., Boudouaia, A., & Zhu, C. (2024). Would ChatGPT-facilitated programming mode impact college students' programming behaviors, performances, and perceptions? An empirical study. *International Journal of Educational Technology in Higher Education, 21*(14). https://doi.org/10.1186/s41239-024-00446-5

Swiecki, Z., Khosravi, H., Chen, G., Martinez-Maldonado, R., Lodge, J. M., Milligan, S., Selwyn, N., & Gašević, D. (2022). Assessment in the age of artificial intelligence. *Computers and Education: Artificial Intelligence, 3*, 100075. https://doi.org/10.1016/j.caeai.2022.100075

Sztipanovits, M., Qian, K., & Fu, X. (2008). The automated web application testing (AWAT) system. *Proceedings of the 46th Annual Southeast Regional Conference on XX*, 88–93. https://doi.org/10.1145/1593105.1593128

Taherkhani, A., Malmi, L., & Korhonen, A. (2008). Algorithm recognition by static analysis and its application in students' submissions assessment. *Proceedings of the 8th International Conference on Computing Education Research*, 88–91. https://doi.org/10.1145/1595356.1595372

Thomson, P. (2021). Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity. *Queue, 19*(4), Pages 10:29-Pages 10:41. https://doi.org/10.1145/3487019.3487021

Thornton, M., Edwards, S. H., Tan, R. P., & Pérez-Quiñones, M. A. (2008). Supporting student-written tests of gui programs. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, 537–541.

https://doi.org/10.1145/1352135.1352316

Tisha, S. M., Oregon, R. A., Baumgartner, G., Alegre, F., & Moreno, J. (2023). An automatic grading system for a high school-level computational thinking course. *Proceedings of the 4th International Workshop on Software Engineering Education for the Next Generation*, 20–27. https://doi.org/10.1145/3528231.3528357

Truong, N., Bancroft, P., & Roe, P. (2003). A web-based environment for learning to program. In *Proceedings of the 26th Australasian Computer Science Conference - Volume 16* (pp. 255–264). Australian Computer Society, Inc.

Truong, N., Roe, P., & Bancroft, P. (2004). Static analysis of students' Java programs. *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30, 30*, 317–325.

Ullah, F., Wang, J., Farhan, M., Jabbar, S., Wu, Z., & Khalid, S. (2020). Plagiarism detection in students' programming assignments based on semantics: Multimedia e-learning based smart assessment methodology. *Multimedia Tools and Applications, 79*(13), 8581–8598. https://doi.org/10.1007/s11042-018-5827-6

Van Verth, P. B. (1985). *A system for automatically grading program quality (metrics, software metrics, program complexity)* [Phd]. State University of New York at Buffalo.

Verma, A., Udhayanan, P., Shankar, R. M., KN, N., & Chakrabarti, S. K. (2021). Source-Code Similarity Measurement: Syntax Tree Fingerprinting for Automated Evaluation. *Proceedings of the First International Conference on AI-ML Systems*, 1–7. https://doi.org/10.1145/3486001.3486228

Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology, 49*(2), 99–107. https://doi.org/10.1016/j.infsof.2006.03.001

Waugh, K., Thomas, P., & Smith, N. (2004, July). Toward the automated assessment of entity-relationship diagrams. In *Second Workshop of the Learning and Teaching Support Network - Information and Computer Science: TLAD (Teaching, Learning and Assessment of Databases)*. https://oro.open.ac.uk/2455/

Wermelinger, M. (2023). Using GitHub Copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)* (pp. 172–178). Association for Computing Machinery. https://doi.org/10.1145/3545945.3569830

Wohlfart, O., & Wagner, I. (2023). Teachers' role in digitalizing education: An umbrella review. *Education Technology Research and Development, 71*(2), 339–365. https://doi.org/10.1007/s11423-022-10166-0

Wut, T., & Xu, J. (2021). Person-to-person interactions in online classroom settings under the impact of COVID-19: A social presence theory perspective. *Asia Pacific Education Review, 22*(3), 371–383. https://doi.org/10.1007/s12564-021-09673-1

Yu, Y. T., Tang, C. M., & Poon, C. K. (2017). Enhancing an automated system for assessment of student programs using the token pattern approach. *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, 406–413. https://doi.org/10.1109/TALE.2017.8252370

Zhao, J., Xia, K., Fu, Y., & Cui, B. (2015). An AST-based Code Plagiarism Detection Algorithm. *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, 178–182. https://doi.org/10.1109/BWCCA.2015.52

Zhu, H. (2012). Position statement: Can software design benefit from creative computing? *2012 IEEE 36th Annual Computer Software and Applications Conference*, 310–311. https://doi.org/10.1109/COMPSAC.2012.119

Appendix A. SUMMARY OF COURSES UTILIZING AUTOMATED ASSESSMENT

Table 2. Summary of courses utilizing automated assessment.

Course name	**Students***	**Demographics****	**Teaching resources*****	**Course content**	**Language**
Computer Programming (6 ECTS)	340-523	Computer Science, Mathematics, Mathematical Statistics, Computer Engineering, Physics, Chemistry & Materials Science & others	1 × Inst. max 18 × TA	Algorithms and programs. Representations of algorithms, flow-charts. Branching algorithms. Loops. Sub-algorithm. Refining algorithms for given text-based problems. Types. Program structure. Names. Variables. Operations. Expressions. Text output. Boolean expressions, comparing. Functions - defining, returning the value, invoking. Conditional statement. Loop statement. One-dimensional arrays. Scanning an array. Returning an array. Nested loops. String processing. String processing. Dictionaries. Tuples. Sets. Input and output. Data exchange with files. Recursion.	Python
Object-oriented Programming (6 ECTS)	241-325	Computer Science, Mathematics, Mathematical Statistics, Computer Engineering & others	1 × Inst. max 12 × TA	Java program, compiling, running Primitive types. Expressions. Selection. Loops. OOP paradigm. Objects and classes. Variable scopes. Eclipse. Static variables. Signature. Java array. Strings. Textual I/O. Inheritance and polymorphism. Overriding. Class Object. Abstract classes and interfaces. Wrapper classes. Interface Comparable. Graphics. JavaFX. Events. Listeners. GUI. Exceptions. Streams. Binary I/O. Dynamical data structures (list, stack, queue). Java data structures. Collection. Interfaces. List and Map. Threads.	Java
Programming II (6 ECTS)	206-309	Computer Science, Mathematics, Mathematical Statistics & others	1 × Inst. max 11 × TA	Arrays. Recursion. Strings.	Java

Course name	**Students***	**Demographics****	**Teaching resources*****	**Course content**	**Language**
Databases (6 ECTS)	231-349	Computer Science, Mathematics, Mathematical Statistics & others	1 × Inst. max 14 × TA	The basic concepts of relational databases. The structure and syntax of SQL. Data description and modification languages. Simple queries, filtering, grouping and aggregation. Views and access restrictions. Data integrity. Database systems and most common data models. The theory of database design in the relational model. Relational languages, their classification, and languages used in practice. Query optimization. Specific questions (data warehouses, data protection, reliability, etc).	SQL
Algorithms and Data Structures (6 ECTS)	195-276	Computer Science, Computer Engineering & others	1 × Inst. max 10 × TA	The concept of data structure. The classical data structures together with related algorithms: sequences, hash tables, trees, graphs.	Java
Automata, Languages and Compilers (6 ECTS)	125-236	Computer Science & others	1 × Inst. max 5 × TA	The course covers introductory topics in compiler construction, focusing on regular expressions, context-free grammars, syntax trees and code generation. Central to this course is one of the most important concepts in computer science: inductive definitions. Context-free grammars is a convenient notation for this. As structural induction is a fundamental proof method in formal verification, understanding the inductively defined structure of computer programs is the first step towards trustworthy program design. Most exercises in this course are designed to develop such an understanding.	Java
Functional Programming (6 ECTS)	41-82	Computer Science & others	1 × Inst. max 4 × TA	Introduction to a FP language. Basic values and operations (including lists). Higher order functions. Types and (parametric) polymorphism. Lazy evaluation and infinite structures. Definition of data types. Untyped lambda calculus. Simply typed lambda calculus. Church's thesis. Type inference. Combinatorial logic. Type classes (ad hoc polymorphism). Monads and input/output. Vectors and other dependent types.	Idris

Course name	Students*	Demographics**	Teaching resources***	Course content	Language
Programming in C++ (6 ECTS)	82-121	Computer Engineering, Computer Science & others	1 × Inst. max 6 × TA	Using GNU C/C++ compiler, Make, Doxygen. Best practice in designing C++ applications. Using the C++ Standard Template Library.	C++
Introduction to Programming (in Estonian) (3 ECTS)	105-192	Geography, Geology & Environmental Technology, Science & Math Teaching, Information Management & others	1 × Inst. max 6 × TA	Algorithms and programs. Representations of algorithms, flow-charts. Branching algorithms. Loops. Sub-algorithms. Developing algorithms for given text-based problems. Program structure. Names. Variables. Operations. Expressions. Boolean expressions, comparisons. Conditional statements. Loop statements. Lists. Functions. User input. Reading from a file. Writing to a file. Simple user interface.	Python
Introduction to Programming II (in Estonian) (3 ECTS)	28-69	Elective course	1 × Inst. max 3 × TA	Algorithms and programs. Representations of algorithms, flow-charts. Branching algorithms. Loops. Sub-algorithm. Refining algorithms for given text-based problems. Number systems. Bit, byte. Types. Program structure. Names. Variables. Operations. Expressions. Text output. Boolean expressions, comparing. Methods, description, return of value, invoke. Conditional statement. Loop statement. One-dimensional arrays. Array scan. Array return. Nested loops. String processing. Input and output. Data exchange with files. Screen graphics. Overview of different programming languages. Main phases of software development.	Python
Introduction to Programming (in English) (3 ECTS)	128-240	IT Law, Science & Technology, Quantitative Economy, Business Administration, Geoinformatics, Innovation & Technology Management & others	1 × Inst. max 6 × TA	Algorithms and programs. Representations of algorithms, flow-charts. Branching algorithms. Loops. Sub-algorithms. Developing algorithms for given text-based problems. Program structure. Names. Variables. Operations. Expressions. Boolean expressions, comparisons. Conditional statements. Loop statements. Lists. Functions. User input. Reading from a file. Writing to a file. Simple user interface.	Python

Course name	**Students***	**Demographics****	**Teaching resources*****	**Course content**	**Language**
Introduction to Programming II (in English) (3 ECTS)	36-100	Elective course	1 × Inst. max 3 × TA	Algorithms and programs. Representations of algorithms, flow-charts. Branching algorithms. Loops. Sub-algorithm. Refining algorithms for given text-based problems. Number systems. Bit, byte. Types. Program structure. Names. Variables. Operations. Expressions. Text output. Boolean expressions, comparing. Methods, description, return of value, invoke. Conditional statement. Loop statement. One-dimensional arrays. Array scan. Array return. Nested loops. String processing. Input and output. Data exchange with files. Screen graphics. Overview of different programming languages. Main phases of software development.	Python

* Number of students varying for the past 5 years.

** Others = Some students who enroll voluntarily. For the rest, it is a mandatory course.

*** Instr. = Responsible instructor. TA = Teaching assistant.

Appendix B. SUMMARY OF MOOCS

Table 3. Summary of MOOCs.

Course name	Students	Demographics	Course content	Language
About Programming (1 ECTS)	Up to 2247	People with no prior experience in programming. The course is designed for individuals of various ages.	Algorithm. Program. Variable. Data types. Conditional statement. Loop. Function. Regular expression. Data handling. Reading and writing files.	Python
Introduction to Programming (3 ECTS)	Up to 1770	Learners with little to no prior experience in programming. Completing the course "About Programming" is beneficial but can be compensated with high motivation and enthusiastic learning.	Algorithm. Program. Variable. Data types. Conditional statement. Loop. Function. Regular expression. Data handling. Reading and writing files. Lists. User interfaces and graphics.	Python
Introduction to Programming II (3 ECTS)	Up to 1047	The course requires programming knowledge at least at the level of the "Introduction to Programming" course.	Two-dimensional lists. Nested loops. Data structures. Referencing and mutation. Data processing with the Pandas module. Recursion. The course concludes with a final project.	Python

Appendix C. INTERVIEW QUESTIONS ON AUTOMATION IN TEACHING AND LEARNING PROCESSES

Introductory questions

These questions aim to explore common routine activities in teaching and the participants' prior experience with automation.

1. **Are there any routine or repetitive tasks in academic work that could benefit from automation?**
    - Are there specific tedious tasks in teaching or administration that you regularly encounter?
    - Could some of these tasks, such as grading, entering grades, or creating/maintaining automated assessments, be automated?
2. **What are your prior experiences with automation in academic work?**
    - Have you been involved in creating, using, or updating any kind of automation (assessment)?

Key questions

These questions focus on identifying specific processes that could use automation for teachers and students.

3. **What aspects of the teaching process should be automated to make instructors' work more efficient or effective?**
    - Are there areas in your subject(s) where automation is most needed?
    - How might automation improve the teaching workflow?
    - Could automation provide instructors with new capabilities, such as:
        - Generating learner profiles?
        - Assigning students to groups based on their profiles?
        - Providing instructors with guidance or actionable insights?
    - Beyond your own courses, what automation opportunities could apply to other courses or disciplines?
4. **What aspects of the student's learning process could be improved or streamlined through automation?**
    - Are there areas in your courses where automation could enhance the learning process?
    - How might automation benefit students' work and academic performance?
    - Could automation provide students with capabilities they currently lack, such as:

    - Personalized feedback?
    - Task management tools?
  - Beyond your own courses, what automation opportunities could apply to the broader student experience across different disciplines?
5. **What would an ideal teaching and learning process look like?**
   - What key elements should be part of an ideal educational process?
   - What activities should be performed by each stakeholder (instructors, students, and administrators) at different stages of the learning process?
   - What should be the role of humans in this process?
   - What should be the role of computers or automated systems?
   - What are the potential risks in designing such an ideal process?
6. **What are your recommendations for the future of automation in teaching and learning?**
   - In which direction should automated systems, such as automated assessment tools, or general teaching automation, progress?
   - What could be the ultimate outcome or goal of such automation initiatives?

Concluding question

This question focuses on the participants' personal involvement in shaping the future vision of automation.

7. **What role do you envision for yourself in implementing this vision?**
   - What are your views on the use of machine learning or artificial intelligence in education?
   - Do you foresee a future where machines replace teachers entirely?
   - Does this possibility evoke concerns or fears?

Appendix D. CATEGORIES, DEFINITIONS, AND ANCHOR EXAMPLES

D.1. Categories, definitions, and anchor examples for automated assessment

Table 4. Categories, definitions, and anchor examples for automated assessment.
Source: https://osf.io/k8vjy

D.2. Categories, definitions, and anchor examples for automated feedback

Table 5. Categories, definitions, and anchor examples for automated feedback.
Source: https://osf.io/8m5cq

D.3. Categories, definitions, and anchor examples for Run I

Table 6. Categories, definitions, and anchor examples for Run I.
Source: https://osf.io/ct5am

D.4. Categories, definitions, and anchor examples for Run II

Table 7. Categories, definitions, and anchor examples for Run II.
Source: https://osf.io/udhtn

Appendix E. PROGRAMMING ASSIGNMENT - DRAWING A FLAG OF AN ESTONIAN RURAL MUNICIPALITY

Task description:
Create a Python program that displays a graphical output in a window with a white background and the title "Flag". The program has to draw the flag of any Estonian rural municipality of your choice (including at least three colors or contain a more complex shape).

The flag must include at least three colors or contain a more complex shape. Everyone makes their own choice. Some quite interesting challenges include the Narva-Jõesuu, Tartu, or Palamuse flags. It is recommended to choose a flag that includes repeating or cyclic elements to make use of loops in your solution.

If desired, you may share an image of your program's output with fellow students in the dedicated forum called "Flag, traffic sign, and house designs".

Note: Instructions for capturing a screenshot are provided. If you have been struggling with the task for a long time, you may seek help from the Troubleshooter, which provides explanations for common issues and useful hints.

Appendix F. OPTIONAL PROGRAMMING ASSIGNMENT - DRAWING ON A FREE TOPIC

Task description:
As part of the mandatory assignment this week, students are required to draw a flag, traffic sign, or house. Many creative solutions have been submitted, and to assess them, an automated evaluation system is used to determine whether the required object is present in the image. While this system generally works well, it is still under development, and there is potential for further improvements.

One area of exploration is testing whether the system can automatically recognize objects that students define themselves. Therefore, in this optional task, students are invited to choose their own topic and create a drawing using Python's Tkinter module.

The submitted program must include the following element in the first line as a comment:

- Exactly one Estonian word describing the object in the drawing (as a comment using #).

Note: Students are encouraged to share their images in the forum titled "Drawing on a free topic - Drawings for other participants", where they can view and discuss each other's work.

Appendix G. QUESTIONS RELEVANT TO GRAPHICAL TASKS FROM RUN I

1. **Please evaluate the complexity of the week 4 assessment tasks.**
   - Task 4.1 (Capital Letters)
   - Task 4.2a (Estonian Administrative Unit Flag)
   - Task 4.2b (Traffic Sign)
   - Task 4.2c (House)

   Each task had to be evaluated on the following scale:
   - 1 - Too easy
   - 2
   - 3
   - 4
   - 5 - Too complicated
   - Didn't solve this task

   1.1 If desired, please comment on your answer.
2. **In this course, we are testing a new automated assessment system for graphical tasks (flag, traffic sign, and house).**
   Please rate the functioning of the automated assessment of graphical solutions.
   - 1 - Worked very poorly
   - 2
   - 3
   - 4
   - 5 - Worked very well
3. **Please provide feedback on the automated assessment of graphical solutions. What are its positive and negative aspects?**
4. **The most interesting topic/task/question was...**
   Write what you found to be the most interesting from the materials of weeks three and four.

Appendix H. QUESTIONS RELEVANT TO GRAPHICAL TASKS FROM RUN II

1. **Which week (1-4) had the most difficult tasks for you?**
   - Week 1
   - Week 2
   - Week 3
   - Week 4

   1.1 If desired, please comment on your answer.
2. **Please rate the functioning of the automated assessment of graphical solutions (flag, traffic sign, and house).**
   - 1 - Worked very poorly
   - 2
   - 3
   - 4
   - 5 - Worked very well
3. **Please provide feedback on the automated assessment of graphical solutions. What are its positive and negative aspects?**
4. **The most interesting topic/task/question was...**
   When answering, consider all four weeks of the course.

Appendix I. CODING SCHEME FOR EVALUATING AUTOMATED ASSESSMENT AND GRAPHICAL TASKS FOR RUN I

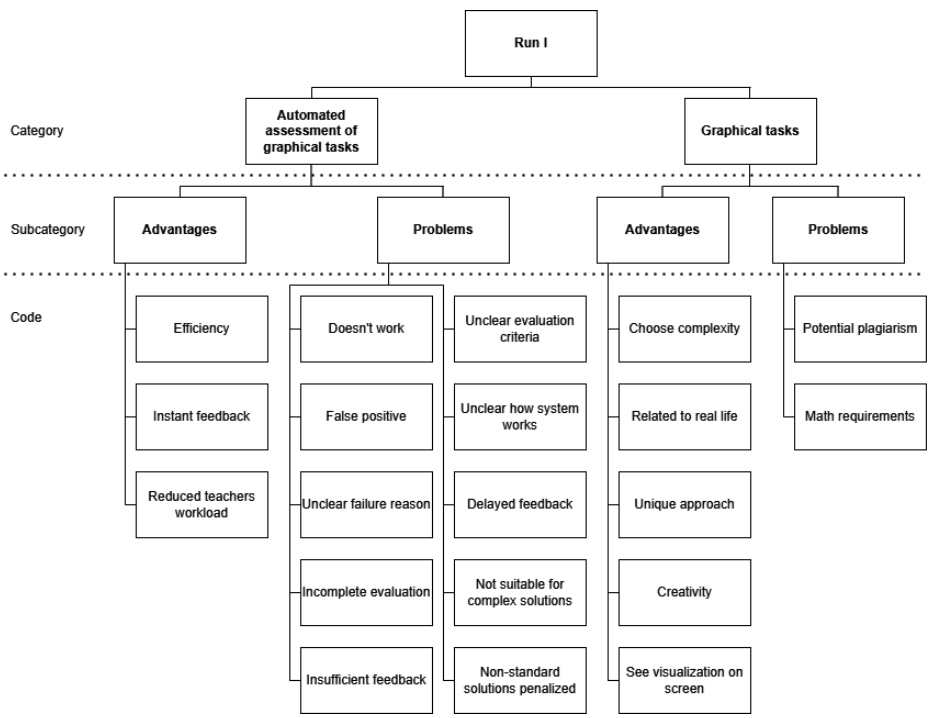

Figure 24. Categories, subcategories, and corresponding codes for evaluating both automated assessment of programming tasks with graphical output and the tasks themselves in Run I.

Appendix J. CODING SCHEME FOR EVALUATING AUTOMATED ASSESSMENT AND GRAPHICAL TASKS FOR RUN II

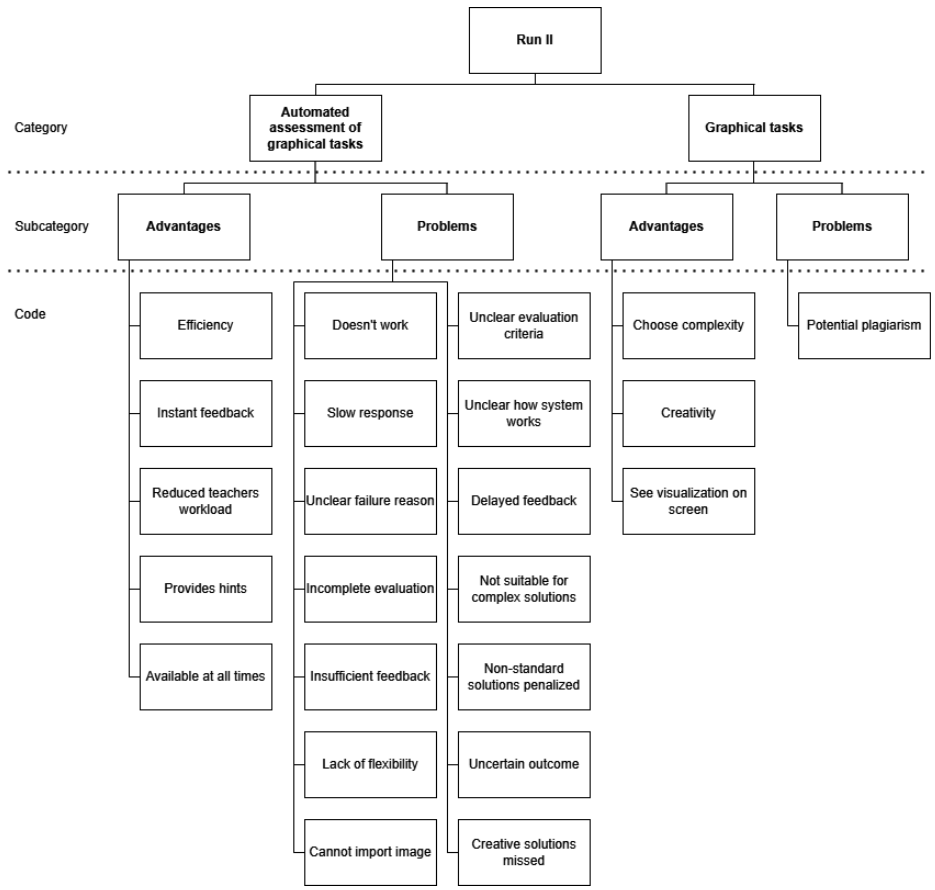

Figure 25. Categories, subcategories, and corresponding codes for evaluating both automated assessment of programming tasks with graphical output and the tasks themselves in Run II.

Appendix K. TSL DOCUMENTATION

K.1. TSL general fields for all the tests

Table 8. TSL general fields for all the tests.

Field	Description	Mandatory
language: String	The programming language this test is meant for. Default value is "python3".	No
validateFiles: Boolean	Condition that decides if basic validations are being executed on the input files.	Yes
requiredFiles: List<String>	The list of files that must be submitted.	Yes
tslVersion: String	The version of TSL to be used.	Yes
tests: List<Test>	The list of tests to be executed (A.3, A.4, A.5).	Yes

K.2. Test general fields for all the different test types

Table 9. Test general fields for all the different test types.

Field	Description	Mandatory
id: Long	Unique test identifier.	Yes
name: String	Name of the test that will be displayed to the user. Default value is test type (A.3, A.4, A.5).	No
pointsWeight: Double	Points weight of the current test. Default value is 1.0.	No
visibleToUser: Boolean	Condition that decides if the user should see this test and its results. Default value is True.	No
inputs: String	Placeholder for displaying the inputs. Currently not in use.	No
passedNext: Long	Placeholder for choosing the next test in case the current test passes. Currently not in use.	No
failedNext: Long	Placeholder for choosing the next test in case the current test fails. Currently not in use.	No

K.3. All the different program test types available in TSL

Table 10. All the different program test types available in TSL.

Test name	Test specific fields	Description
program_execution_test	standardInputData: List<String> inputFiles: List<FileData> genericChecks: List<GenericCheck> outputFileChecks: List<OutputFileCheck> exceptionCheck: ExceptionCheck	Program execution test.
program_contains_try_except_test	programContainsTryExcept: ContainsCheck	Test that verifies if a program contains a try/except block.
program_calls_print_test	programCallsPrint: CallsCheck	Test that verifies if a program calls the print function.
program_contains_loop_test	programContainsLoop: ContainsCheck	Test that verifies if a program contains a for/while loop.
program_imports_module_test	genericCheck: GenericCheckLong	Test that verifies if a program imports specific module(s).
program_contains_keyword_test	genericCheck: GenericCheck	Test that verifies if a program contains specific keyword(s).
program_calls_function_test	genericCheck: GenericCheckLong	Test that verifies if a program calls specific function(s).
program_defines_function_test	genericCheck: GenericCheckLong	Test that verifies if a program defines specific function(s).
program_defines_class_test	genericCheck: GenericCheckLong	Test that verifies if a program defines specific class(es).
program_defines_subclass_test	className: String superClass: String beforeMessage: String passedMessage: String failedMessage: String	Test that verifies if a program defines specific subclass(es).
program_calls_class_test	genericCheck: GenericCheckLong	Test that verifies if a program calls specific class(es).
program_calls_class_function_test	genericCheck: GenericCheckLong	Test that verifies if a program calls specific class function(s).

K.4. All the different function test types available in TSL

Table 11. All the different function test types available in TSL.

Test name	Test specific fields	Description
function_execution_test	functionName: String functionType: FunctionType createObject: String arguments: List<String> standardInputData: List<String> inputFiles: List<FileData> genericChecks: List<GenericCheck> returnValueCheck: ReturnValueCheck paramValueChecks: List<ParamValueCheck> outputFileChecks: List<OutputFileCheck> outOfInputsErrorMsg: String functionNotDefinedErrorMsg: String tooManyArgumentsProvidedErrorMsg: String	Function execution test.
function_contains_loop_test	functionName: String containsLoop: ContainsCheck	Test that verifies if a function contains a for/while loop.
function_contains_keyword_test	functionName: String genericCheck: GenericCheck	Test that verifies if a function contains specific keyword(s).
function_contains_return_test	functionName: String containsReturn: ContainsCheck	Test that verifies if a function contains the return keyword.
function_calls_function_test	functionName: String genericCheck: GenericCheckLong	Test that verifies if a function calls specific function(s).
function_calls_print_test	functionName: String callsCheck: CallsCheck	Test that verifies if a function calls the print function.
function_is_recursive_test	functionName: String isRecursive: RecursiveCheck	Test that verifies if a function is recursive.
function_defines_function_test	functionName: String genericCheck: GenericCheckLong	Test that verifies if a function defines specific function(s).
function_imports_module_test	functionName: String genericCheck: GenericCheckLong	Test that verifies if a function imports specific module(s).
function_contains_try_except_test	functionName: String containsTryExcept: ContainsCheck	Test that verifies if a function contains try/except block.
function_is_pure_test	functionName: String containsLocalVars: ContainsCheck	Test that verifies if a function contains only local variables.

K.5. All the different class test types available in TSL

Table 12. All the different class test types available in TSL.

Test name	Test specific fields	Description
class_imports_module_test	val className: String genericCheck: GenericCheckLong	Test that verifies if a class imports specific module(s).
class_defines_function_test	val className: String genericCheck: GenericCheckLong	Test that verifies if a class defines specific function(s).
class_calls_class_test	val className: String genericCheck: GenericCheckLong	Test that verifies if a class calls specific class(es).
class_function_calls_function_test	val className: String classFunctionName: String genericCheck: GenericCheckLong	Test that verifies if a class function calls specific function(s).
class_instance_test	val className: String classInstanceChecks: List<ClassInstanceCheck> createObject: String	Test that verifies if a class instance has specific attributes with specific values after specific operations.

K.6. TSL test fields

Table 13. TSL test fields.

Field	Description	Mandatory
standardInputData	The input that the user can enter via the keyboard.	No
inputFiles	List of input files that will be generated by the test consisting of file name and content.	No
genericCheck	A specific check that allows to validate different conditions within a given string.	No
outputFileChecks	A specific check that allows to validate different conditions within a given file.	No
exceptionCheck	Check that verifies if an exception is thrown.	No
containsLoop	Check that verifies if a while/for loop is present.	Yes
containsReturn	Check that verifies if a return statement is present.	Yes
callsCheck	Check that verifies if the print function is called.	Yes
isRecursive	Check that verifies if a function is recursive.	Yes
programContainsTryExcept	Check that verifies if a try/except block is present.	Yes
programCallsPrint	Check that verifies if the print function is called.	Yes
programContainsLoop	Check that verifies if a while/for loop is present.	Yes
containsLocalVars	Check that verifies if a function consists of only local variables.	Yes
containsTryExcept	Check that verifies if a try/except block is present.	Yes
className	Class name.	Yes
classFunctionName	Class function name.	Yes
classInstanceChecks	Check that verifies if a class instance contains specific fields with specific values.	No
superClass	Superclass name.	Yes
createObject	A block of Python code that is executed within the test to set custom content/context for the test.	No
beforeMessage	A message to be displayed to the user before the execution of the test.	Yes
passedMessage	A message to be displayed to the user in case the test passes.	Yes
failedMessage	A message to be displayed to the user in case the test fails.	Yes
functionType	Enum: Function/Method.	Yes
arguments	List of arguments for the given function.	No
returnValueCheck	Check that verifies if a function returns the correct value.	Yes
paramValueChecks	List of function parameters with specific index and value to be checked for a given function.	No

Field	Description	Mandatory
outOfInputsErrorMsg	Custom error message for the case where the program asks more user inputs than were given.	No
functionNotDefinedErrorMsg	Custom error message for the case where the testable function is not defined in the solution code.	No
tooManyArgumentsProvidedErrorMsg	Custom error message for the case where the given function expects a different number of arguments than was given.	No

K.7. TSL objects descriptions

Table 14. TSL objects descriptions.

Field	Sub-fields	Description
FileData	fileName: String fileContent: String	Describes the content of the file with a given name.
GenericCheck	checkType: CheckType nothingElse: Boolean expectedValue: List<String> elementsOrdered: Boolean dataCategory: DataCategory ignoreCase: Boolean	A specific check that allows to validate different conditions within a given string.
GenericCheckLong	checkType: CheckType nothingElse: Boolean expectedValue: List<String> elementsOrdered: Boolean dataCategory: DataCategory ignoreCase: Boolean	A specific check that allows to validate different conditions within a given string.
OutputFileCheck	fileName: String checkType: CheckType nothingElse: Boolean expectedValue: List<String> elementsOrdered: Boolean dataCategory: DataCategory ignoreCase: Boolean	A specific check that allows to validate different conditions within a given file.
ExceptionCheck	mustNotThrowException: Boolean	Check type that verifies if an exception is thrown.
ContainsCheck	mustNotContain: Boolean	Check type that verifies if a condition is met.
CallsCheck	mustNotCall: Boolean	Check type that verifies if a specific function is called.
FunctionType		Enum: Function/Method.
ReturnValueCheck	returnValue: String	Check type that verifies if a function returns the correct value.
ParamValueCheck	paramNumber: Int expectedValue: String	Check type that verifies if the given function parameter is correct on a specific position.
RecursiveCheck	mustNotBeRecursive: Boolean	Check type that verifies if a specific function is recursive.
ClassInstanceCheck	fieldsFinal: List<FieldData> checkName: Boolean checkValue: Boolean nothingElse: Boolean	Check type that verifies if a class instance has given fields with a given name and a given value.
FieldData	fieldName: String fieldContent: String	Describes the content of the field with a given name.

K.8. TSL object CheckType and CheckTypeLong field values

Table 15. TSL object CheckType and CheckTypeLong field values.

Field	Description
ALL_OF_THESE	Checks if all the elements provided are found in the target list of elements.
ANY_OF_THESE	Checks if any of the elements provided are found in the target list of elements.
ANY	Checks if there are any elements in the target list of elements.
NONE_OF_THESE	Checks if none of the elements provided are found in the target list of elements.
MISSING_AT_LEAST_ONE_OF_THESE	Checks if at least one of the elements provided is found in the target list of elements.
NONE	Checks if there are no elements in the target list of elements.

K.9. TSL object DataCategory field values

Table 16. TSL object DataCategory field values.

Field	Description
CONTAINS_LINES	From the given input, it extracts lines using newline characters and removes any whitespace before or after the line.
CONTAINS_NUMBERS	From the given input, it extracts all the numeric values.
CONTAINS_STRINGS	From the given input, it extracts all the occurrences of the given strings.
EQUALS	Simply removes all the whitespace before and after the given input.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisors.

To Eno Tõnisson – your support meant the world to me. You were not just a supervisor, but a true mentor and a father figure. Our conversations always started with sports, news, or something completely off-topic, creating a warm and safe space even during the hardest moments. Your passing left a hole in this journey, but your influence has carried me through to the finish line. This thesis is as much yours as it is mine.

To Marina Lepp – thank you for believing in me, especially when I didn't believe in myself. You knew exactly how to lift me up when I was down and helped us build a rhythm and teamwork that made the impossible feel achievable. I'm immensely grateful for your unwavering support.

I also want to thank Reimo Palm, who, in many ways, acted like a third supervisor. Your attention to detail and care for the technical aspects of the work often exceeded even my own.

My sincere thanks go to the Didactics of Informatics research group. The writing camps, group events, and shared struggles with other PhD candidates were invaluable. Knowing others were facing similar battles helped me push through. Special thanks to Tauno and Marili, who made me laugh when I needed it most and reminded me not to take everything so seriously.

To the Lahendus team – Kaspar, Merka, and Priit – thank you for the guidance and collaboration during the integration and development phases. You helped me make thoughtful decisions toward building something sustainable and usable. I also truly enjoyed our casual chats about life, which often provided much-needed balance.

To my family, thank you from the bottom of my heart. To my wife Mariliis – your unconditional support and patience throughout these years means everything to me. Thank you for helping me maintain balance between work, studies, and family, and for always being the steady force that kept me grounded. To my parents, brothers, and sister – thank you for always encouraging me to take care of myself, and for cheering me on as I approached the finish line.

To my friends – thank you for keeping me grounded and sane. Lauri, sharing this path with you has been a real highlight – that class we shared will always be unforgettable. Markus, thank you for always being there and rolling the uniques together (sending Pogo luck). Kevin and Ermo, your efforts to keep me physically active and relaxed made a real difference — whether it was chasing that elusive 100kg bench or making sure we had proper R6 taku time to reset the brain. Somehow, both were equally therapeutic. Big shoutouts to team Discgolf (Igor is still unbeaten Legend), team Ironmen, team Padel (one day I will smash it), and team Pamela – and all the other friends who stood by me.

The degree study has been financially supported by the Institute of Computer Science at the University of Tartu and IT Academy.

SISUKOKKUVÕTE

IT õpetamise hindamis- ja tagasisideprotsesside automatiseerimine – loomise ja haldamise täiustamine õppejõudude vaatenurgast

Infotehnoloogia (IT) valdkond on viimaste aastakümnetega kasvanud ning see on endaga kaasa toonud märkimisväärse nõudluse kvalifitseeritud spetsialistide järele. Vabade töökohtade arv on omakorda suurendanud IT-hariduse ja sellega seotud karjääride poole pürgivate õppijate arvu. Erinevad õppeasutused ning platformid, alustades lasteaedadest, veebipõhistest platvormidest ning lõpetades ülikoolidega, proovivad pakkuda erinevaid võimalusi IT-alaste teadmiste omandamiseks. Olgugi, et selline trend on tööstuse jaoks kasulik, toob see haridusasutustele kaasa mitmeid väljakutseid. Üheks peamiseks murekohaks on tekkinud vajadus hallata suuremaid gruppe õppureid, säilitades piisavat kvaliteeti pakutud haridusele, näiteks andes õppijatele õigeaegset ning individuaalset tagasisidet. Traditsioonilised hindamise ning tagasisidestamise meetodid pole suurenenud hulga üliõpilaste, ülesannete ning kursuste tõttu enam paljudele õpetajatele erinevates õpetamiskeskkondades ning -vormides jätkusuutlikud. See doktoritöö uurib hindamis- ja tagasisideprotsesside automatiseerimise võimalusi IT-hariduses, keskendudes programmeerimiskursustele, kus ülesannete hindamine ning tagasisidestamine on sageli korduv ning aeganõudev protsess.

Esimene panus doktoritöös on välja selgitada, millised on suurimad probleemid ning väljakutsed programmeerimiskursustel ning kuidas automatiseerimine võiks aidata neid probleeme lahendada. Doktoritöös kasutatakse kvalitatiivset lähenemist, viies õppejõududega intervjuusid läbi minigruppides. Intervjuud tõid esile mitmeid olulisi protsesse, mida saaks automatiseerida või mille olemasolevat automatiseerimist tuleks täiustada. Ühe olulise puudusena tuli välja, et graafilise väljundiga programmeerimisülesannetel ei ole automaatkontrolle ning nende hindamine on ajamahukas, sest lisaks koodi läbivaatamisele tuleb see ka käivitada, et valideerida graafilist väljundit. Samuti selgus intervjuudest, et olemasolevad automaatkontrollid ei ole õppejõududele piisavalt kasutajasõbralikud ega paindlikud, mistõttu nende kasutamine ning haldamine nõuab liigset tehnilist oskust. Nende intervjuude tulemused andsid selge suuna edasistele uurimis- ja arendustegevustele.

Teine panus doktoritöös on pildituvastusel põhineva automaatse hindamissüsteemi loomine, mis võimaldab hinnata graafilise väljundiga programmeerimisülesannete lahendusi. Loodud süsteem genereerib üliõpilase esitatud koodist graafilise väljundi ning esitab selle Clarifai-nimelisele teenusepakkujale, kes saadab vastusena tõenäosuse, et soovitud objekt esineb antud graafilisel väljundil. Selle tõenäosuse järgi määratakse ülesanne kas arvestatuks või mittearvestatuks. Süsteemi rakendati mitmetel programmeerimiskursustel, kus see vähendas oluliselt õppejõudude töömahtu ning ühtlasi andis üliõpilastele kohest tagasisidet. Kursus-

tel osalejatelt küsisti ka tagasisidet ning nad hindasid enim süsteemi võimekust pakkuda kiiret ja täpset tagasisidet, mis võimaldas neil oma lahendusi vajadusel täiustada.

Kolmas panus doktoritöös on TSL-i (Test Specific Language) loomine ning rakendamine. TSL-i eesmärk on lihtsustada ja standardiseerida automaatkontrollide loomist ning haldamist. TSL võimaldab defineerida ühe ülesande automaatseks hindamiseks kõik kriteeriumid struktureeritud ning lihtsal viisil. TSL-i rakendamiseks loodi TSL-i parsija ning kompilaator, mis valideerivad ning teisendavad selle hindamisskriptiks. Automaatseks hindamiseks loodi teek Tiivad, mis käivitab genereeritud skripti, teostab automaatkontrollid ning tagastab tulemused. Selleks, et TSL-i ei tuleks luua käsitsi, on loodud kasutajaliides, mis lihtsustab õppejõududel testide seadistamist ning haldamist. Kõik need komponendid töötavad koos, et vähendada õppejõudude töökoormust, võimaldades neil kiiresti ja täpselt määratleda hindamiskriteeriumid. Kasutajaliides kaotab vajaduse sügavate tehniliste teadmiste järele. Loodud süsteem standardiseerib ja kiirendab hindamisprotsessi, tehes selle lihtsamaks ja vähem aeganõudvaks, võimaldades õppejõududel keskenduda rohkem õpetamise sisulistele aspektidele.

Kokkuvõttes annab doktoritöö olulise panuse IT-hariduse valdkonda, pakkudes lahendusi, mis võimaldavad automatiseerida hindamis- ja tagasisideprotsesse ning vähendada õppejõudude töökoormust. Loodud süsteemid mitte ainult ei paranda hindamiste kiirust ja täpsust, vaid pakuvad ka skaleerimisvõimalusi suurematele kursustele. Töö tulemused ja loodud süsteemid aitavad suunata tulevasi uurimusi ja arendustegevusi IT-hariduses, pakkudes tuge tõhusamate ja paindlikumate hindamissüsteemide väljatöötamiseks.

PUBLICATIONS

CURRICULUM VITAE

Personal data

Name:	Eerik Muuli
Date of birth:	21.02.1993
Citizenship:	Estonia
Contact:	eerik.muuli@ut.ee

Education

2019–2025	University of Tartu, Doctoral degree (PhD), Computer Science
2015–2017	University of Tartu, Master's degree (MSc), Software Engineering
2012–2015	University of Tartu, Bachelor's degree (BSc), Computer Science
2009–2012	Hugo Treffner Gymnasium
2000-2009	Luunja Secondary School

Employment

2023–...	Microsoft, Senior Software Engineer
2021–2023	Microsoft, Software Engineer
2017–2021	STACC (Software Technology and Applications Competence Center), Software Developer
2014–2017	Playtech Estonia, Client QA Engineer
2013–2014	Koolisüsteemide OÜ, Application Administrator

Scientific work

Main fields of interest:

- Use of computers in learning and teaching computer science
- Informatics didactics

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Eerik Muuli
Sünniaeg:	21.02.1993
Kodakondsus:	Eesti
Kontakt:	eerik.muuli@ut.ee

Haridus

2019–2025	Tartu Ülikool, Doktorikraad (PhD), Informaatika
2015–2017	Tartu Ülikool, Magistrikraad (MSc), Tarkvaratehnika
2012–2015	Tartu Ülikool, Bakalaureusekraad (BSc), Informaatika
2009–2012	Hugo Treffneri Gümnaasium
2000-2009	Luunja Keskkool

Teenistuskäik

2023–...	Microsoft, Vanem-Tarkvaraarendaja
2021–2023	Microsoft, Tarkvaraarendaja
2017–2021	STACC (Software Technology and Applications Competence Center), Tarkvaraarendaja
2014–2017	Playtech Estonia, Kliendi Kvaliteedi Tagamise Insener
2013–2014	Koolisüsteemide OÜ, Rakenduse Administraator

Teadustegevus

Peamised uurimisvaldkonnad:

- Arvutite kasutamine informaatika õppimisel ja õpetamisel
- Informaatika didaktika

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.
30. **Hina Anwar.** Towards Greener Software Engineering Using Software Analytics. Tartu 2021, 186 p.
31. **Veronika Plotnikova.** FIN-DM: A Data Mining Process for the Financial Services. Tartu 2021, 197 p.
32. **Manuel Camargo.** Automated Discovery of Business Process Simulation Models From Event Logs: A Hybrid Process Mining and Deep Learning Approach. Tartu 2021, 130 p.
33. **Volodymyr Leno.** Robotic Process Mining: Accelerating the Adoption of Robotic Process Automation. Tartu 2021, 119 p.
34. **Kristjan Krips.** Privacy and Coercion-Resistance in Voting. Tartu 2022, 173 p.
35. **Elizaveta Yankovskaya.** Quality Estimation through Attention. Tartu 2022, 115 p.
36. **Mubashar Iqbal.** Reference Framework for Managing Security Risks Using Blockchain. Tartu 2022, 203 p.
37. **Jakob Mass.** Process Management for Internet of Mobile Things. Tartu 2022, 151 p.
38. **Gamal Elkoumy.** Privacy-Enhancing Technologies for Business Process Mining. Tartu 2022, 135 p.
39. **Lidia Feklistova.** Learners of an Introductory Programming MOOC: Background Variables, Engagement Patterns and Performance. Tartu 2022, 151 p.
40. **Mohamed Ragab.** Bench-Ranking: A Prescriptive Analysis Approach for Large Knowledge Graphs Query Workloads. Tartu 2022, 158 p.
41. **Mohammad Anagreh.** Privacy-Preserving Parallel Computations for Graph Problems. Tartu 2023, 181 p.
42. **Rahul Goel.** Mining Social Well-being Using Mobile Data. Tartu 2023, 104 p.

43. **Anti Ingel.** Algorithms using information theory: classification in brain-computer interfaces and characterising reinforcement-learning agents. Tartu 2023, 142 p.
44. **Shakshi Sharma.** Fighting Misinformation in the Digital Age: A Comprehensive Strategy for Characterizing, Identifying, and Mitigating Misinformation on Online Social Media Platforms. Tartu 2023, 158 p.
45. **Kristiina Rahkema.** Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects. Tartu 2023, 182 p.
46. **Ivan Slobozhan.** Studying Online Social Media Engagement in CIS Countries during Protests, Mass Demonstrations and War. Tartu 2023, 81 p.
47. **Nurlan Kerimov.** Building a catalogue of molecular quantitative trait loci to interpret complex trait associations. Tartu 2023, 248 p.
48. **Pavlo Tertychnyi.** Machine Learning Methods for Anti-Money Laundering Monitoring. Tartu 2023, 117 p.
49. **Abasi-amefon Obot Affia.** A Framework and Teaching Approach for IoT Security Risk Management. Tartu 2023, 180 p.
50. **Raimond-Hendrik Tunnel.** Video Game Design and Development Bachelor's Curriculum for Estonia. Tartu 2024, 137 p.
51. **Ahto Salumets.** Bioinformatics analysis of various aspects in immunology. Tartu 2024, 198 p.
52. **Mohammed Abdulhameed Shaif Ali.** Deep Learning Methods for Cell Microscopy Image Analysis. Tartu 2024, 143 p.
53. **Pille Pullonen-Raudvere**. Foundations of Efficient and Secure Algorithm Development for Secure Multiparty Computation. Tartu 2024, 265 p.
54. **Marili Rõõm.** Multiple approaches to learners' success and factors affecting it in computer programming MOOCs. Tartu 2024, 170 p.
55. **Shivananda Rangappa Poojara.** Design and Orchestration of Scalable, Event-Driven Serverless Data Pipelines for Internet of Things (IoT) Applications. Tartu 2024, 172 p.
56. **Hassan Abdulgaleel Hassan Salim Eldeeb.** Empowering Machine Learning Pipelines with Automated Feature Engineering. Tartu 2024, 121 p.
57. **Muhammad Uzair.** Soft decision making for agri-food 4.0. Tartu 2024, 158 p.
58. **Kirill Milintsevich.** Estimation of Depression Level from Text: Symptom-Based Approach, External Knowledge, Dataset Validity. Tartu 2024, 130 p.
59. **Maksym Del.** Multilingual and Multi-Domain Representational Patterns Across Trpansformer-Based Models. Tartu 2024, 131 p.
60. **Kristo Raun.** Adaptive Out-of-order Handling in Streaming Conformance Checking. Tartu 2024, 118 p.
61. **Toivo Vajakas.** Towards integration of mobile network data into analyzing human mobility. Tartu 2024, 103 p.
62. **Katsiaryna Lashkevich.** Data-Driven Analysis and Optimization of Waiting Times in Business Processes. Tartu 2024, 169 p.
63. **Alejandra Duque-Torres.** Classifying, Constraining and Ranking Metamorphic Relations. Tartu 2025, 159 p.

64. **Mariia Bakhtina.** A Method for Information Security and Privacy Management in Smart Solutions. Tartu 2025, 199 p.
65. **Andre Tättar.** Multilingual Machine Translation for Under-Resourced Languages. Tartu 2025, 170 p.
66. **Mahmoud Shoush.** Prescriptive Process Monitoring Under Uncertainty and Resource Constraints. Tartu 2025, 178 p.
67. **Alireza Akhavi Zadegan.** A Multimodal approach for refining Mapping and Localization by Integrating Generative AI and Pedestrian-Centric Data. Tartu 2025, 147 p.