DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
75

HEIDI TAVETER

Using Programming-Process Data of Introductory Programming Courses: Finding Solver Types, Giving Feedback, and Detecting Plagiarism

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
75

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
75

HEIDI TAVETER

Using Programming-Process Data of Introductory Programming Courses: Finding Solver Types, Giving Feedback, and Detecting Plagiarism

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 December 9, 2025 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

Assoc. Prof. Marina Lepp
University of Tartu
Estonia

Assoc. Prof. Eno Tõnisson †
University of Tartu
Estonia

Opponents

Prof. Erik Barendsen
Radboud University
the Netherlands

Assoc. Prof. Julien Broisin
University of Toulouse
France

The public defense will take place on January 20, 2026 at 11:00 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 978-9908-57-099-0 (print) ISBN 978-9908-57-100-3 (pdf)

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

To my family and friends

ABSTRACT

Over time, programming has become a crucial skill that students from various backgrounds and disciplines increasingly learn. Since much of the learning takes place outside the traditional classroom setting, teaching methods must be adapted, and additional tools are needed to support these courses effectively. Varying levels of prior programming experience among students further increase the diversity, resulting in a notably wide range of educational needs. Furthermore, high dropout rates remain a significant issue in introductory programming courses. The early weeks of an introductory programming course are especially critical, as they most indicate students' eventual success. Given all these factors, it is essential to explore how students differ in their learning approaches while programming and identify effective ways to support them, particularly during the crucial initial phase of the course.

This thesis aims to explore how programming-process data can be utilized to profile and support students in introductory programming courses and to identify instances of plagiarism. Log data was collected from different introductory programming courses taught at the University of Tartu, such as "Computer programming", "Introduction to programming", "Introduction to programming II", and "From Technology Consumer to Creator". The data was utilized to identify and analyze different solver types, examine their variation and consistency over time, provide log-based feedback, evaluate its effect on student performance, and find relevant plagiarism detection characteristics based on programming process information. Data was also collected from a pre-course questionnaire on prior programming experience, from the Moodle learning management system and the Study Information System. The main approach was quantitative. Clustering was used as the primary method for forming and analyzing solver types. An experiment was conducted to get information about the efficacy of the log-based feedback.

This research demonstrated that students can be grouped into distinct solver types based on their behavior patterns in programming, with similar patterns appearing among both beginners and non-beginners. The main solver types identified are: (1) "Frequent pressers of the run button", (2) "Receivers of syntax errors", (3) "Balanced solvers", and (4) "Late starters of the program execution". A notable discovery was identifying a new behavior feature in programming associated with lower performance, which is the late start of the first execution of the program. Balanced solvers with below-average values across all features can be linked to higher performance, while some groups achieved comparable results despite differences in behavior patterns, emphasizing the diversity. It is essential to note that compared to the first midterm exam, groups had more notable differences in the second midterm scores. This may also suggest that skill gaps widen as the course progresses. The study also revealed that solver types are not persistent during an introductory programming course. Furthermore, the study

showed that feedback derived from programming log data significantly improved exam test results, which focused on code reading skills. The experiment revealed that log-based feedback notably decreased the time beginners needed to complete programming tasks and increased their exam test scores. Programming-process data also proved valuable for enhancing plagiarism detection methods. Specifically, the study found that general style features, due to their consistency, could be effective in history-based plagiarism detection tools. This history-based approach gave input for developing the tool Thonny Log Analyzer, which compares students' programming-process data for plagiarism detection.

Based on the results of this thesis, an important recommendation is to promote a teaching approach that encourages students to run their programs frequently as a regular and integral part of the coding process. Educators can show the best practices in practice sessions by demonstrating program writing habits that use regular executions. The results also underscore the importance of focusing on debugging skills in the early weeks of the course. Focusing on debugging can help decrease the number of students who choose not to run their programs. According to the findings of the thesis, it is effective to use log-based feedback regularly in programming courses to improve beginners' code-reading abilities. It is essential, especially considering that in the era of AI, it is increasingly necessary to develop strong code-reading skills.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CP	University course "Computer Programming"
IDE	Integrated development environment
IP1	University course "Introduction to Programming"
IP2	University course "Introduction to Programming II"
IPM	MOOC "Introduction to Programming"
MOOC	Massive open online course
SIS	Study Information System
TCC	Course "From Technology Consumer to Creator" for high-school students
VPL	Virtual Programming Lab (Moodle plugin)

LIST OF ORIGINAL PUBLICATIONS

Publications Included in the Thesis

I **Meier, H.**, E. Tõnisson, M. Lepp, and P. Luik (2020). "Behaviour Patterns of Learners while Solving a Programming Task: an Analysis of Log Files". In: *2020 IEEE Global Engineering Education Conference (EDUCON)*. Vol. 11. IEEE, pp. 685–690. DOI: https://doi.org/10.1109/educon45650.2020.9125134.

II **Meier, H.** and M. Lepp (2021). "Style Features in the Programming Process Which Can Help Indicate Plagiarism". In: *7th International Conference on Higher Education Advances (HEAd'21)*. Vol. 7. Editorial Universitat Politècnica de València, pp. 623–630. DOI: https://doi.org/10.4995/head21.2021.13072.

III **Meier, H.** and M. Lepp (2023a). "Effectiveness of Feedback Based on Log File Analysis in Introductory Programming Courses". In: *Journal of Educational Computing Research* 61.3, pp. 696–719. DOI: https://doi.org/10.1177/07356331221132651.

IV **Meier, H.** and M. Lepp (2023b). "Clusters of Solvers' Behavioral Patterns Based on Analysis of the Programming Process". In: *2023 IEEE Frontiers in Education Conference (FIE)*. Vol. 53. IEEE, pp. 1–6. DOI: https://doi.org/10.1109/FIE58773.2023.10343479.

V **Meier, H.**, M. Lepp, and R. Kütt (2024). "Plagiarism Detection Tool Based on Programming Activity Logs". In: *2024 IEEE Global Engineering Education Conference (EDUCON)*. Vol. 15. IEEE, pp. 1–7. DOI: https://doi.org/10.1109/educon60312.2024.10578885.

VI **Taveter, H.** and M. Lepp (2025). "Clusters of Solvers' Behavioral Patterns Among Beginners and Non-beginners and Their Changes During an Introductory Programming Course". In: *Informatics in Education* 24.1, pp. 199–221. DOI: https://doi.org/10.15388/infedu.2025.07.

Author's Contribution to the Publications

Publications I–IV, VI	Formulating the research questions, data collection, data analysis, literature review, and writing the paper as the main author.
Publication V	Supervising with the second author the tool that was developed by the third author, writing sections III–V.

Note on the Usage of AI

I acknowledge using ChatGPT (4o mini) to reformulate some parts of my text in chapters 3–6 and for identifying synonyms.

1. INTRODUCTION

This chapter outlines the research problem that motivated the author to explore how data from the programming process can be utilized to profile and support students in introductory programming courses and detect plagiarism. The chapter concludes by presenting three research questions of the study.

1.1. Research Problem

Programming has, over time, become an essential skill increasingly learned by students from different backgrounds and disciplines (Luxton-Reilly et al., 2018; O'Malley & Aggarwal, 2020). The changing situation has brought several challenges to programming education. One is large courses, where online learning has a more significant role. So, the courses need adapted teaching methodology and additional tools for supporting students (Castro & Tumibay, 2021). It is important to consider that much of learning occurs outside the classroom. Still, the need for support can be different depending on the student (Santos et al., 2013; Song et al., 2021). The second challenge is increasing diversity. The situation where there are large courses and, in addition, more students with non-major contexts creates a situation in which students' educational needs are relatively varied (O'Malley & Aggarwal, 2020). The students' different previous programming experiences further increase the variability. For example, beginners have to do more work to solve the programming tasks (Vihavainen et al., 2014b). In addition to the challenges already described, high dropout rates remain a problem in introductory programming courses (Bennedsen & Caspersen, 2019; Luxton-Reilly, 2016; Medeiros et al., 2018; Watson & Li, 2014). From this perspective, the importance of the first few weeks of an introductory programming course is emphasized. Research has revealed that the third and fourth weeks are most predictive of final outcomes (Porter & Zingaro, 2014), and some authors have named the first weeks in the same context (Ahadi et al., 2014; Estey & Coady, 2016). On the other hand, another study has found that the best prediction can be made based on the weeks in the middle of the course (Chen et al., 2022). Therefore, considering all the aspects mentioned above, it is essential to find ways to understand the differences between students in programming and how we can better support their learning, especially at the very beginning of the programming course.

One of the ways to gain knowledge about students' behavior in programming is to use programming-process data. Data collected about the programming process can be analyzed and used for various purposes. Among them are student profiling for detecting behavior patterns in programming, giving feedback, and possibilities for plagiarism detection. Behavior patterns in programming can be detected using behavior features in programming. In this thesis, "behavior feature in programming" means a variable that can be captured using recorded data from logs. Another essential term in this thesis, "behavior pattern in programming",

is a particular way of acting during the programming process, including at least two behavior features in programming. Third term, "solver type", means in this thesis a group characterized by a particular behavior pattern in programming. The terms "pattern of programming behaviors" (Pereira et al., 2020), "characteristics of programming behavior" (Watson et al., 2013), "behavior" (Vihavainen et al., 2014b), "programming behaviour," "behaviour pattern" (Hosseini et al., 2014), "programming pattern" (Blikstein et al., 2014), "coding pattern" (Fujiwara et al., 2012) have also been used in a similar sense.

Several studies have pointed out that students' behavior patterns in programming vary (Hosseini et al., 2014; Pereira et al., 2020; Vihavainen et al., 2014a). Some studies have used environments that record detailed information about the programming process, which has helped to understand beginners' behavior patterns in programming. For example, it has been found that beginners copy and paste a lot at the beginning of the course (Blikstein, 2011; Vihavainen et al., 2014a), and they use a lot of trial-and-error attempts (Blikstein, 2011; Hosseini et al., 2014). At the same time, research has also revealed that some students never execute their programs (Carter & Hundhausen, 2017). Significant differences can sometimes indicate a peculiarity on which the results do not depend (Blikstein et al., 2014). However, some behavior patterns in programming indicate difficulties or inefficiencies that can be addressed to improve outcomes. For example, higher-performing students have better skills of dealing with errors (Pereira et al., 2020; Watson et al., 2013; Zhang et al., 2023b), and they usually change their code more between submissions (Pereira et al., 2020). However, in previous studies, profiling has not primarily addressed the gradual, stepwise nature of the programming process, and there remains room for further integration of essential features in programming into profiling. Because the findings are partially inconsistent, it is necessary to examine more closely which behavior patterns in programming are related to the performance. Finding behavior patterns in programming that indicate higher and lower performance can provide input to develop ways to improve the course. Moreover, there is limited research on behavior patterns in programming that investigates beginners and non-beginners as distinct groups and uses more detailed data than submission level.

Log data is also a valuable resource for getting information to use for giving feedback to students. For example, based on logging data of the programming environment, the knowledge of students' recent errors has been used for group work and practicing debugging (Deeb et al., 2018). The possibility is also applied so learners can analyze their programming process after solving the task (Matsuzawa et al., 2013). Some programming-process visualization tools are created so teachers can effectively analyze where students struggle and then discuss problem-solving techniques or use other possibilities to support students in a classroom (Rubinstein et al., 2019; Yan et al., 2019; Zhang et al., 2023a). In addition, efforts have been made to find ways to use detailed programming-process data to analyze when and how to provide automatic feedback (Jeuring

et al., 2022). For example, the research proposes instructions that include automated questions for learners who exhibit trial-and-error behavior and provide positive feedback when they complete a subgoal. Considering the above, it can be said that giving feedback based on log data provides possibilities to focus on improving process-related skills in addition to current programming topics, and it is important to find the best practices for it. It is essential to note that the practice of integrating programming-process-based feedback with teaching in a classroom is relatively recent. Furthermore, only a few studies address feedback to enhance programming-process-specific skills and students' abilities to read and interpret programs.

Using programming-process data to detect plagiarism is becoming increasingly essential. One reason is that tools based on comparative source code analysis are ineffective against obfuscation methods such as reordering statements, renaming variable names, etc. (Anjali et al., 2015; Herrera et al., 2019; Novak et al., 2019; Rodríguez-Veliz et al., 2023). In addition, plagiarism detection is especially complicated in introductory courses because programs are relatively simple and short (Modiba et al., 2016; Ryman et al., 2021; Ryman et al., 2022). At the same time, most plagiarism detection tools are still based on source code analysis (Li et al., 2023; Ljubovic & Pajic, 2020; Novak et al., 2019). However, steps have also been taken toward using data from the programming process to check and deter plagiarism. For this purpose, publicly available IDE plugins that generate logs with keystroke-level data have been used (Hart et al., 2023; Hellas et al., 2017). The history of repository commits has also been used for plagiarism checking (Rodriguez-Rivera et al., 2022). One functionality for getting information about potential issues is replaying the programming process and visualizing it (Shrestha et al., 2022) or generating graphs about significant changes, for example, insertions and deletions (Rodriguez-Rivera et al., 2022). In addition, some studies use timing information of keyboard press and release events during programming (Byun et al., 2020). Other features can also be valuable for detecting academic dishonesty, for example, average or maximum length of paste, number of pastes, number of compilations, number of successful compilations, coding speed, number of short and long pauses, etc. (Ljubovic & Pajic, 2020). Finding possibilities to develop effective plagiarism detection tools based on programming-process data can make it more challenging to use obfuscation methods and can help teaching, especially in large courses. Although progress has been made, using tools based on programming-process analysis remains relatively time-consuming as each student's work must be reviewed individually. Consequently, there is a need for tools that can perform more of the comparative analysis automatically and flag cases that require further inspection.

1.2. The Focus of the Research

This thesis aims to determine how programming-process data can be used to profile and support students in introductory programming courses and check plagiarism.

The thesis addresses the following research questions:

- RQ1: What types of solvers can be differentiated from the analysis of the programming process, in which statistically significant differences exist in midterm exam scores between solver types, and how persistent are these types over time (the whole group, beginners, and non-beginners)?
  Research question 1 is addressed in Publications I, IV and VI.
- RQ2: How does feedback based on logs affect exam results, task completion time, number of runs, error messages, and pastes (of the whole group, beginners, and non-beginners)?
  Research question 2 is addressed in Publication III.
- RQ3: What features based on programming-process data can be used to detect plagiarism?
  Research question 3 is addressed in Publications II and V.

2. THEORETICAL BACKGROUND

This chapter provides an overview of the theoretical basis for this thesis. In the beginning, various behavior patterns in programming are described, followed by a discussion of the possibilities of student profiling. Then, the relationship between behavior patterns and performance and the influence of previous programming experience is observed. In addition, using information from the programming process to provide feedback to students and check plagiarism is considered.

2.1. Solvers' Behavior Features in Programming

This section looks at how the study of the programming process has shown that students' behavior patterns in programming differ in various ways, and studying the variations helps to understand what students do during the programming. Behavior patterns in programming have been studied using data of varied degrees of detail and different behavior features in programming (Hundhausen et al., 2017). For example, some works are based on information from submissions (e.g., Albluwi & Salter, 2020), some use copies of student code every time they compile their programs (e.g., Tabanao et al., 2011), while others save all students' actions, including keystrokes and all code changes (e.g., Blikstein, 2011). Different degrees of detail can be identified, but the main data levels can be divided as follows (coarsest to finest): (1) submission level, (2) snapshot level and (3) keystroke level (Villamor, 2020). Using programming-process data, in points of view, it is necessary to consider that about half of the students work with programs they never submit, and some submit their work and continue working with the same programs (Vihavainen et al., 2014b). Using a system that recorded a series of snapshots, it has been found that students make more conceptual changes during the first quartile of task solving, and then it reduces quickly (Hosseini et al., 2014). In addition, students write different amounts of code between compilations or saves (Ardimento et al., 2022), and the amount and time of deleted code varies (Hosseini et al., 2014). Some students use a lot of copy-pasting, and some do not (Blikstein, 2011; Vihavainen et al., 2014a). Most students add code gradually, while some add a large part of code and then start modifying it (Hosseini et al., 2014). Some students use a lot of trial-and-error attempts (Blikstein, 2011; Jemmali et al., 2020; López-Pernas & Saqr 2021; Michaeli & Romeike, 2019), while others try to write the program without running it (Ardimento et al., 2022; Carter & Hundhausen, 2017). Students also take pauses of varying lengths during programming (Leinonen et al., 2022; Shrestha et al., 2022). Some students use debugging to fix errors, while most do not use it (Ardimento et al., 2022). Interestingly, it has been revealed that the students who make more edits in code work more in a programming environment without dealing with materials outside of the editor (Leinonen et al., 2017). It is also essential to note that sometimes students act differently in different phases of the programming process, and then

we can analyze which behavior is dominant (Hosseini et al., 2014). For example, sometimes a student solves only one part of the assignment incrementally.

Although different programming languages are used in introductory programming courses, e.g., Python (Pereira et al., 2020; Shrestha et al., 2022), Java (Hosseini et al., 2014; Leinonen et al., 2022; Vihavainen et al., 2014a), NetLogo (Blikstein, 2011), and C++ (Bey & Champagnat, 2022; Carter & Hundhausen, 2017), the general trends observed in programming behaviour are remarkably similar across different languages. Attempts have also been made to compare languages. For example, using a data-driven approach, it has been found that students who use Python in an introductory programming course receive fewer error messages, require fewer attempts, and spend less time compared to those using Java (Lokkila et al., 2023; Naveed, 2024). In addition, Python programs are shorter. At the same time, the language does not influence the most common error types students receive (Lokkila et al., 2023). Another study found context-dependent differences in digraphs; however, typing speed and the possibility to use digraphs in distinguishing students are not influenced by context (Edwards et al., 2020).

In summary, research has shown that it is possible to detect a wide variation of behavior patterns in programming based on different granularity levels of data. However, there is still little research on the programming process at a level more detailed than submissions.

2.2. Student Profiling for Finding Solver Types

One direction in the analysis of the programming process is student profiling. This has been done using different analysis options and varied behavior features in programming. This subsection discusses the different methods for student profiling to find solver types. Student profiling also helps to find connections between behavior patterns in programming and students' performance (e.g., Pereira et al., 2020; Piech et al., 2012; Shrestha et al., 2022; Zhang et al., 2023b). These connections are discussed in subsection 2.3.

One way of student profiling is to do it based on detailed programming-process data but form solver types manually using the main behavior feature as the basis of dividing (e.g., Blikstein, 2011; Hosseini et al., 2014). Some researchers have used machine learning methods, such as cluster analysis. Clustering can be based on a single behavior feature in programming, such as pauses (Shrestha et al., 2022) or debugging (Zhang et al., 2023b), or include many different behavior features in programming related to the programming process (Pereira et al., 2020). When dividing students into groups, bottom-up and top-down strategies are also distinguished (Shi et al., 2023). The bottom-up approach means that students are divided into groups based on behavior features in programming, for example, through cluster analysis (e.g., Bey et al., 2019, Pereira et al., 2020). The top-down strategy implies that, at first, students are divided based on characteristics such as performance. Then, differences and similarities in behavior patterns

in programming are compared within pre-fixed groups (e.g., Albluwi & Salter, 2020).

Different researchers have formed groups, which will be discussed below. Based on the small number of students but detailed information on code insertions, deletions, compilations and error messages and their timeline monitoring, three main groups have been formed: “Copy and pasters”, “Mixed-mode” and “Self-sufficient” (Blikstein, 2011). While the code of students who belong to the “Self-sufficient” group grows linearly, containing pauses, “Copy and pasters” sometimes use the previous solution as a starting point or copy a large amount of code during programming. The “Mixed-mode” group works in a way that is a combination of the previous two. Somewhat similar characteristics and detailed individual-level information are also the basis of a study where students were divided into four groups: "Builders", "Massagers", "Reducers", and "Strugglers" (Hosseini et al., 2014). “Builders” work incrementally, and correctness also grows gradually. “Massagers” have periods when they make small code changes without visible progress. “Reducers” reduce concepts during the programming. “Strugglers” have problems to write correct code for a long time. Students have also been divided by the number of unit test runs into three groups, which were named “Intellects”, “Thinkers”, and “Probers” (Sharma et al., 2018). "Intellects" ran tests the lowest number of times, had more code changes between two tests than others and had the best score in the first trial. “Thinkers” ran tests more frequently than “Intellects” but less frequently than “Probers”. The amount of their code changes was also between two other groups. "Probers" ran tests most frequently, had the most minor code changes between two tests and had a similar score in the first trial than "Thinkers".

Listed below are some studies where machine learning methods have been used to form groups. Using cluster analysis and snapshots in every compilation, students have been grouped based on the development path (Piech et al., 2012). The first group of students make steadily small steps towards the correct solution. The other group of students sometimes had moments when they encountered serious functional problems, and they made big steps from a semi-working program to a working correct program. Other research used cluster analysis and data about keystrokes, compile/run events, and focused on pauses (Shrestha et al., 2022). Students were divided into longer pause and shorter pause clusters. Cluster analysis has also been used to investigate differences in finding and fixing errors (Zhang et al., 2023b). Group A students were good and fast in debugging. Group B students suffered from logic and runtime errors, while group C had problems mainly with syntax errors. There have also been studies where several behavior features in programming, such as the number of submissions, the average time between two submissions, the average number of changes and the percentage of submissions with syntactical errors, have been included in the cluster analysis (Bey et al., 2019). Cluster 1 students had many submissions with brief gaps between submissions, and their code changes were irregular. Cluster 2 students had a small

number of submissions with long time gaps and the largest code changes between submissions. Cluster 3 had the lowest number of submissions, with brief gaps and minor changes between submissions. Cluster 1 students had more syntax errors than Cluster 2, but there were no essential differences between clusters. Another work, which used behavior features in programming related to submissions, divided students into six clusters (Bey & Champagnat, 2022). Cluster 1 students were good at composing the solution but encountered syntax errors because they were hurrying. Cluster 2 members moved between different solutions and changed their code profoundly between submissions. Cluster 3 represents students who had difficulties compiling the solution. Cluster 4 students were overall good because they took time to conceive their solution. Cluster 5 students experienced difficulties composing the solutions because they spent too little time on them, and Cluster 6 members submitted their code frequently to get feedback and learn from their mistakes. There is also an example of research that used cluster analysis and many different behavior features in programming based on more fine-grained data (Pereira et al., 2020). It allowed for information on various aspects of the programming process. Error messages received during programming, repeated error messages, correctness, the ratio between pasted and typed characters, and code changes between submissions are included in this study. Features related to the use of the programming environment, such as procrastination, time spent in IDE, number of logins, etc., have also been added. Cluster A students had the lowest number of error messages, repeated error messages, and the lowest ratio between pasted and typed characters. Still, they spent the most time in IDE and had the biggest number of logins into the environment. Cluster C students had the highest number of error messages, repeated error messages, and the highest ratio between pasted and typed characters. Still, they spent the least time in IDE and had the lowest number of logins into the environment. Cluster B was between the previous two regarding several behavior features in programming, but they had the shortest time between starting programming and the assignment deadline.

In summary, various methods of student profiling have been found, from manual dividing based on the main feature in programming with a detailed process description to cluster analysis based on multiple features in programming. Still, in most works, the profiling focus has not been on graduality in the programming process, and there are still possibilities to incorporate new essential features as part of profiling.

2.3. Relations between Behavior Features in Programming and Performance

The variations in the programming process have led researchers to ask which behavior features indicate good performance in programming, and which indicate difficulty or inefficiency. Error messages have been widely considered an important feature related to performance (Bey et al., 2019; Carter et al., 2015; Estey

& Coady, 2016; Jadud, 2006; Pereira et al., 2020). It has been found that students with fewer error messages perform better (Tabanao et al., 2011). It has been pointed out separately that well-performing students have fewer syntax errors (Pereira et al., 2020). At the same time, among better-performing students are also those who are good at finding solutions but make syntax errors because they hurry to get their assessment results (Bey & Champagnat, 2022). In addition, attention has been paid to the speed and efficiency of error correction. It has been revealed that students who can deal with errors faster perform better (Pereira et al., 2020; Watson et al., 2013; Zhang et al., 2023b). Concerning errors, it has also been observed that students who perform worse do not have a systematic approach to programming but use a trial-and-error method (Heinonen et al., 2014). However, there are studies that highlight that some students do not check code correctness at all (Ardimento et al., 2022; Carter & Hundhausen, 2017), and those who do not execute programs are more likely to fail the exam (Carter & Hundhausen, 2017). The last fact also points to the importance of debugging. It is emphasized that good debugging skills are the door to better results (Zhang et al., 2023b). The studies have also found that systematic teaching of debugging skills increases the efficiency and speed of finding bugs (Alqadi, 2024). Interestingly, the students who get average results use a debugger the most (Carter & Hundhausen, 2017). The students with the best outcome use a debugger with the same frequency as the students who get worse results. However, the results are contradictory. Namely, a study found that the number of times a debugger was used was negatively correlated with program correctness (Leinonen et al., 2017).

In addition to the features mentioned above, it has been revealed that copy-paste usage (Pereira et al., 2020) and a greater number of long pauses (with duration over ten minutes) (Shrestha et al., 2022) are indications of lower results. The latter feature can mean that the students who have more frequent long pauses use that time to search for information from materials. The frequency of submissions, changes, and the time between them has also been studied. It has been revealed that weaker students submit more frequently (Albluwi & Salter, 2020; Bey et al., 2019) and make fewer code changes between submissions (Pereira et al., 2020). It aligns with the finding that students who start submitting while only some lines are written perform worse (Bey & Champagnat, 2022). It is essential to note that this is the case only if we use submission-level data. Another study investigated code update frequency and size based on snapshots, which are related to execution times, and did not find correlations with course outcomes (Blikstein et al., 2014). In addition, it has also been revealed that faster students tend to have fewer saves and executions (Hosseini et al., 2014). Attention has also been paid to how changes in behavior patterns in programming during the course are related to student progress. Interestingly, based on the code update size and frequency, it has been revealed that students whose behavior patterns in programming change more during the course perform better (Blikstein et al., 2014). It aligned with other work that showed that at-risk students can be detected in the first two weeks

of the course and their behavior remains the same during the semester (Estey & Coady, 2016). Some studies have looked at learners' programming time over a whole semester. Learners who achieve better results have been found to spend more time in the programming environment (Munson & Zitovsky, 2018; Pereira et al., 2020). The amount of programming time at home and the final result of the course are particularly strongly related (Munson & Zitovsky, 2018). Consequently, the students with better results practice more. It was also observed that high-achieving students tended to use more independent strategies from the beginning of the course, as evidenced by their minimal use of help-seeking sessions, and they maintained this approach throughout the course (López-Pernas & Saqr, 2021).

In conclusion, different behavior patterns in programming are related to student performance, but the results are partly contradictory. It is necessary to study more thoroughly which ineffective behavior patterns in programming indicate a lack of necessary skills that could be improved by focusing more on them in teaching, and which patterns simply characterize beginners or non-beginners.

2.4. Influence of Previous Programming Experience on Behavior Patterns in Programming and Performance

The behavior patterns in programming have also been studied from the perspective of how they relate to students' previous experiences or lack of these. For example, based on keystroke-level data, learners have been found to copy and paste a lot at the beginning of their first programming course when they do not have any previous experience yet (Vihavainen et al., 2014a; Blikstein, 2011). Beginners copy code from previous assignments they have solved or from examples in the materials. They also use their previous programs as a starting point for the next programs (Blikstein, 2011). The use of a trial-and-error approach (Blikstein, 2011; Jemmali et al., 2020) and difficulties in finding syntax errors (Denny et al., 2012; Marceau et al., 2011) are also associated with beginners. Beginners also have more errors in initial submissions (Albluwi & Salter, 2020). Based on the above, beginners need different help than more experienced learners, and it depends on their behavior patterns in programming. For example, if they use copy-pasting, they need more examples to learn from and use in solving programming tasks (Blikstein, 2011). It should also be taken into account that the workload of beginners in solving programming tasks is much bigger than that of those with some previous experience (Vihavainen et al., 2014b). The differences between beginners' and experienced students' behavior patterns in programming can be used for different purposes. This can be taken into account when giving feedback. Namely, it is revealed that the usefulness and effectiveness of feedback can depend on the experience level of learners (Worsley & Blikstein, 2013). Also, systems that exploit differences in behavior patterns in programming to distinguish beginners from non-beginners can be used to work out support strategies (Leinonen et al., 2016).

It is essential to note that research has revealed a correlation between course outcomes and previous programming experience (O'Malley & Aggarwal, 2020; Veerasamy et al., 2018; Zhang et al., 2013). It has been considered in many studies that deal with predicting student achievement (Hellas et al., 2018). Research has shown that students with prior programming experience get better results on midterm and final exams (O'Malley & Aggarwal, 2020). Some contexts, such as object-oriented programming and teaching methodologies, may reduce the outcome differences (Ventura & Ramamurthy, 2004). Interestingly, it has been revealed that previous programming experience is an essential predictor of the first programming course results but does not have the same significant influence on the outcomes of subsequent programming courses (Holden & Weeden, 2003). Also, students with previous programming experience are more self-efficient before the programming course, and this remains the case after the course (Wiedenbeck et al., 2004).

In sum, the behavior patterns in programming and performance of beginners and non-beginners differ in introductory programming courses. So, it is necessary to consider this when studying student behavior patterns in programming and their connections with performance. In addition, there is little research related to student behavior patterns in programming that has examined beginners and non-beginners separately.

2.5. Feedback Related to the Programming Process

Feedback, its quality, and opportunities in programming courses for beginners continue to be important research topics (Rocha et al., 2023). Early feedback at the beginning of learning programming has also been emphasized, as it improves the effectiveness of students' approaches and reduces the use of trial-and-error strategy (Ebrahimi et al., 2012). Also, it has been found that feedback is effective if it is concise and instructive (Fu et al., 2023). Feedback can be classified into distinct categories based on different aspects. A widely used classification includes: (1) knowledge of performance, which reports summative feedback, for example, the proportion of tasks completed correctly, grade, and number of errors; (2) knowledge of results, which indicates whether the learner's response is correct or incorrect, or focuses on pointing out mistakes; and (3) knowledge of the correct result, which presents or explains the correct answer (Keuning et al., 2018; Narciss, 2008; Narciss, 2013; Narciss & Huth, 2006). While the previous classification described simple feedback, the following five types represent elaborated feedback elements, each targeting a specific aspect of the instructional context: (1) knowledge about task constraints, which offers guidance on task requirements and general suggestions on how to approach the task; (2) knowledge about concepts, which offers hints as conceptual clarifications and illustrative examples related to the subject matter; (3) knowledge about mistakes, which indicates different types of information about mistakes, for example, test failures, locations and types of er-

rors, some details about logic errors and style issues; (4) knowledge about how to proceed, which can include hints for error correction, information about the next steps, potential problem-solving strategies, or various types of improvements; and (5) knowledge about meta-cognition, which concerns, for example, a learner's awareness of the most appropriate strategy for solving a given problem, checking progress while solving, or evaluating their approach, also offering information on the strategies required for self-regulation of the learning process (Keuning et al., 2018; Narciss, 2008; Narciss, 2013).

The ability to collect detailed information about the programming process provides new opportunities for giving feedback. One way is to create tools that help learners analyze their programming process. For example, a tool has been created that allows one to replay the programming process and see automatically generated results of metrics such as solving time, the number of compilations and runs, etc. (Matsuzawa et al., 2013). The feedback showed that students rated the tool highly because of its usefulness in analyzing the programming process. An option has also been used in which students are sometimes required to verbally reflect on, for example, error messages while solving programming tasks (Deeb & Hickey, 2021). In this case, they can improve the code after sending reflections about plans to fix the error, and teachers can see the reflective comments written by students. The effect was that students solved tasks quicker and had fewer error messages, but on the other hand, more students gave up. Some efforts have been made to generate automatic example-based feedback when students request help during the programming process (Zhi et al., 2019). They use this option because research has revealed that beginners need examples to learn. In addition to the feedback related to different errors, motivational messages are also effectively used to increase students' engagement (Holanda et al., 2023; Marwan et al., 2020).

It has been considered essential to create resources to support the interaction between students and teachers in the classroom. For example, a tool has been developed that visualizes the programming process and provides feedback to teachers and students so that teachers and students can talk together about problem-solving techniques (Yan et al., 2019). The aim was to achieve meaningful integration of automated feedback with the personal feedback from teachers. There are also examples of systems that help inspect and monitor students' steps in real-time in the classroom (Rubinstein et al., 2019; Yu et al., 2023; Zhang et al., 2023a). Systems typically have views of varying degrees of generalization, but they differ in focus. Some of them have a view of misconceptions (Rubinstein et al., 2019), but in some cases the teacher sees a real-time view of each learner's activities, which are placed on the dashboard to reach more learners than teachers could manage in person (Guo, 2015). Some environments include real-time information on students' programming behavior at both general and individual levels so teachers can adjust observed knowledge instantly in classroom teaching (Yu et al., 2023; Zhang et al., 2023a). The overviews a teacher can see about their class

provide diverse added value. For example, these have been found to help quickly identify struggling students, analyze their behavior, and assess the need to give additional feedback (Zhang et al., 2023a). Based on real-time information that covers individual and class-wide tendencies, teachers can adapt their instructions more precisely to the needs of students. They can discuss mistakes they observed in many working processes, give tailored feedback and offer exercises that improve relevant skills based on the observed information (Zhang et al., 2023a). It has been found that a good combination of online feedback with generalization adjusted in classroom teaching is effective (Yu et al., 2023).

In conclusion, several ways have been found to improve feedback using programming-process data. The approach of combining information from programming-process data and teaching in a classroom is still relatively new. In addition, only a few studies focus on feedback research to improve the programming-process-specific skills and the ability to read and interpret programs.

2.6. Using Programming-Process Data in Plagiarism Detection

Programming-process data is valuable in plagiarism detection, especially in the context of introductory courses. Specifically, first programs tend to be simple and short, making it difficult to detect dishonesty because the code could be similar even when created independently (Modiba et al., 2016; Ryman et al., 2021; Ryman et al., 2022). In addition, students use different obfuscation methods to mask their plagiarism. Typically, they change variable names, change the order of variables, make differences in comments, reorder program parts or statements in program parts, add nonrequired code lines, spread code over several lines, etc. (Đurić & Gašević, 2013; Novak et al., 2019). Although there are various algorithms developed for the detection of source-code plagiarism, including those based on style, semantics, text, attribute counting, structure, string matching, etc. (Denzler et al., 2024; Hrkút et al., 2023; Novak et al., 2019), comparison of source codes is ineffective against obfuscation methods (Herrera et al., 2019; Novak et al., 2019; Rodríguez-Veliz et al., 2023). However, it is essential to mention that with the increasing prevalence of AI-generated code, comparing styles has become more important again, for example, by counting style anomalies (Denzler et al., 2024). It is a possible direction in source code comparison and using programming-process data as well.

Among the newer plagiarism-checking approaches are those that use programming-process data. For example, some IDEs have plugins for logging keystroke-level data (Hart et al., 2023; Hellas et al., 2017). Some systems add logged information to a local file, and if teachers decide to use logs for plagiarism checks, students must submit logs with their solutions (Hart et al., 2023). Then, teachers have the possibility to replay the programming process. Although it is possible to detect plagiarism, it is quite time-consuming to replay all students' programming

processes separately if a student has renamed variables, shuffled lines of code, copied large amounts of code, or retyped plagiarized code (Hart et al., 2023).

Also, tools have been created that in addition to replaying the programming process provide some visualization (Shrestha et al., 2022). This is designed to enable an immediate overview of the general characteristics of the programming process; additionally, it is possible to replay the process and view the final solution simultaneously. Visualization of code-changing patterns can help detect plagiarized solutions more effectively, but every programming process needs to be checked separately (Shrestha et al., 2022). Some web-based environments enable the use of keyboard press and release events to increase automation in potential plagiarism detection (Byun et al., 2020) or use the average time it takes to type character pairs — digraphs (Longi et al., 2015). It is more common in a natural language context but there have also been attempts to develop and use it in the programming-process context in introductory programming courses.

In addition to keystroke-level data, there are other options for using programming-process data for plagiarism detection. For example, the repository history data can be used for checking plagiarism (Koss & Ford, 2013; Rodriguez-Rivera et al., 2022). The systems provide different statistics, such as the number of commits over time and tests passed, or create cumulative graphs based on inserted and deleted lines of code (Rodriguez-Rivera et al., 2022). For example, a graph that has mostly inserted lines and very few deleted ones would indicate a potential case of dishonesty. Tools can also detect who starts early and who procrastinates (Koss & Ford, 2013). In addition, some well-known systems can be used to check the originality of each submitted version (Rodriguez-Rivera et al., 2022; Schleimer et al., 2003).

In approaches to plagiarism detection, steps have been taken to incorporate data from the programming process. Using tools based on programming-process analysis is still quite time-consuming. Even solutions that use visualizations usually require quite a lot of work from teachers because of the need to check every student's work separately. Therefore, there is a need for tools that make more comparative analysis automatically and detect cases that require additional checks.

3. METHODOLOGY

This chapter provides an overview of the research methodology. In the beginning, the research design is presented, followed by an introduction to the context of the study and a description of the sample and data collection process. Finally, data analysis methods are explained.

3.1. Research Design

The following research model (Figure 1) illustrates the directions to explore based on students' programming-process data and the potential connections with performance and previous programming experience. The presented potential connections are based on the literature review.

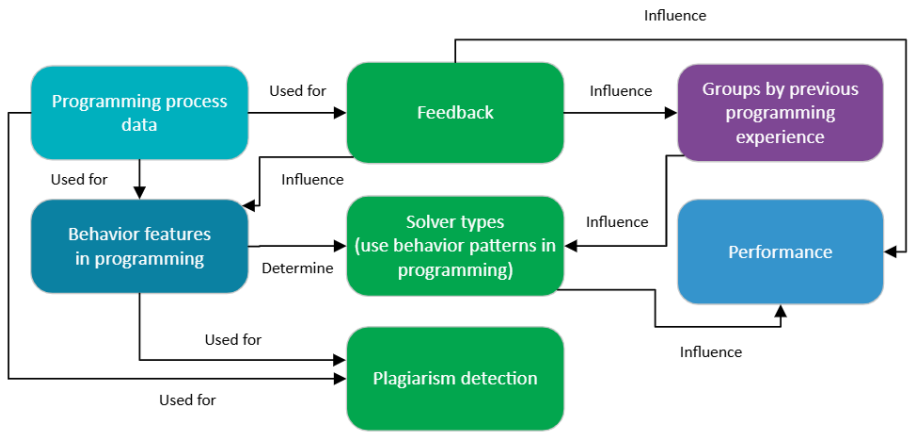

Figure 1. Research model for the doctoral study

Three main directions in using programming-process data are student profiling, giving feedback, and plagiarism detection. Student profiling is based on behavior features in programming. It offers possibilities to detect different behavior patterns in programming, and it is essential to consider the possible influence of previous programming experience and connections with performance. Performance is also related to receiving feedback. For plagiarism detection, programming-process data can be used in different ways, including using behavior features in programming. In this research, the programming-process data are obtained from the programming environment Thonny, which logs user actions during programming. More detailed information about Thonny is given in context subsection 3.2.

The three research questions were addressed based on the three mentioned ways to apply programming-process data to support students in introductory courses and check plagiarism. The research questions and corresponding samples, data sources and data analysis methodology are presented in Table 1.

Table 1. Research questions, samples, data sources, and analysis methods

Research questions	Sample	Data sources	Data analysis methods
RQ1: What types of solvers can be differentiated from the analysis of the programming process in which statistically significant differences exist in midterm exam scores between solver types, and how persistent are these types over time (the whole group, beginners, and non-beginners)?	MOOC "Introduction to Programming", year 2017, 27 learners University course "Computer Programming", year 2022: • 301 students for the first midterm exam • 275 for the second midterm exam • 233 students who participated in both midterm-exams	**Thonny logs** (behavior features in programming) **Pre-questionnaire** (previous experience) **Moodle** (midterm exam scores) **Study Information System** (background information)	**Z-score** to standardize the scores on the same scale **K-means cluster analysis** to find solver types **Kruskal-Wallis test** to evaluate the existence of any significant differences between clusters **Mann-Whitney U test** to compare features pairwise between clusters **Descriptive statistics** of performance
RQ2: How does feedback based on logs affect exam results, task completion time, number of runs, error messages, and pastes (of the whole group, beginners, and non-beginners)?	University course "Introduction to Programming", the years 2020 and 2021 • Experimental group: 25 students • Control group: 55 students	**Thonny logs** (behavior features in programming, programming-process information) **Moodle** (midterm exam scores) **Pre-questionnaire** (previous experience) **Study Information System** (background information)	**Chi-Square test** to check the balance of groups **Mann-Whitney U test** to evaluate the existence of any significant differences between groups **Descriptive statistics** of performance and behavior features in programming
RQ3: What features based on programming-process data can be used to detect plagiarism?	University courses "Introduction to Programming" and "Introduction to Programming II", year 2020, 17 students For tool evaluation: "From Technology Consumer to Creator" for high-school students, year 2023, 141 participants University course "Introduction to Programming", year 2022, 44 students	**Thonny logs** (behavior features in programming)	**Descriptive statistics** of general style features and programming style features

3.2. Context of the Study

This subsection introduces the courses from which data were collected for the study or where the experiment was conducted. Considering the different aspects of the research, five programming courses taught at the University of Tartu are related to the research: "Computer Programming" (hereafter CP), "Introduction to Programming" (hereafter IP1), "Introduction to Programming II" (hereafter IP2), MOOC "Introduction to Programming" (hereafter IPM), and "From Technology Consumer to Creator" (hereafter TCC). All five (CP, IP1, IP2, TCC, IPM) are introductory programming courses in Python and are taught in Estonian. The courses differ in type, target group, form of study, volume, etc., described in Table 2. University courses (CP, IP1, IP2) use flipped classroom methodology. It means that students first familiarize themselves with each new topic at home, solve programming tasks, and answer test questions. Then, they deepen their knowledge of the same topic in the classroom under the guidance of a teaching assistant and get feedback. In the years 2020 and 2021, during the pandemic, the practical sessions were conducted on Zoom. Exams for all courses include test questions and programming tasks to be solved under controlled conditions. The test questions mainly test the program reading skills, and the programming tasks evaluate the ability to write programs. The task instructions describe in detail the functionality of the program(s) to be created and the parts of the program(s). The program(s) must be created within a certain time. Exams in the courses last 90-180 minutes (see Table 2 for more details), of which a maximum of 30 minutes is allocated to answering test questions. All courses required the use of the Thonny programming environment and the submission of Thonny logs during the course. Submission of Thonny logs is also mandatory when students are writing programming tasks in the exam. Thonny is used in the Python introductory courses because it is a Python programming environment developed specially for beginners' courses (Annamaa, 2015). Its functionalities help students who struggle at the beginning of learning programming. For example, it has the option to debug programs, which allows for effective detection of mistakes and learning how programs work. Thonny also has functionalities of replaying the programming process and logging user actions in detail during the programming processes. These user actions (saved with timestamps) encompass activities such as editing the program text by distinguishing between pasted and manually typed text, interacting with standard input, output, and error streams (stdin, stdout, stderr), loading and saving files, executing programs, utilizing stepping commands, etc. (Annamaa, 2015). Both last-mentioned options — replaying the programming process and logging user actions — also provide opportunities for educational research and plagiarism detection. CP is compulsory for most students, IP1 and IP2 are mandatory for some curricula, but students can choose them as elective courses regardless of their study level.

Table 2. Courses from which data were collected

	Computer Programming (CP)	**Introduction to Programming (IP1)**	**Introduction to Programming II (IP2)**	**From Technology Consumer to Creator (TCC)**	**Introduction to Programming (IPM)**
Name in Estonian	Programmeerimine	Programmeerimise alused	Programmeerimise alused II	Tehnoloogia tarbijast loojaks	Programmeerimise alused
Type of course	University course	University course	University course	Course for schools	MOOC
Target group	Students studying computer science as a major, some groups for other specialties	Students who are not studying computer science as a major	Students who are not studying computer science as a major	Learners of age group 16–25	Learners of different ages, not only students
Forms of study	Blended learning	Blended learning	Blended learning	Online learning	Online learning
Duration	16 weeks	7 weeks	7 weeks	10 weeks	8 weeks
Volume	6 ECTS* (156h)	3 ECTS (78h)	3 ECTS (78h)	3 ECTS (78h)	3 ECTS (78h)
Main topics	Variables, conditional statements, loops, functions, reading from files, writing to files, graphics, nested loops, recursion, object-oriented programming, and data structures: lists, sets, dictionaries, and tuples	Variables, conditional statements, loops, lists, functions, reading from files, writing to files	Nested loops, recursion, graphics, object-oriented programming, and data structures: lists, sets, dictionaries, and tuples	Variables, conditional statements, loops, lists, functions, reading from files, writing to files	Variables, conditional statements, loops, lists, functions, reading from files, writing to files
Assessment	Two midterm exams and a final exam (grading scale A–F)	Exam (pass/fail)	Exam (grading scale A–F)	Exam (pass/fail)	Exam (pass/fail)
Exam tasks	In midterm exams: test questions (pass/fail), two programming tasks (max score 20 points). In exam: test questions (pass/fail), three programming tasks (max score 30 points)	Test questions (max score 24 points), a programming task (max score 24 points)	Test questions (max score 26 points), two programming tasks (max score 26 points)	Test questions (max score 40 points), a programming task (max score 40 points)	Test questions (max score 40 points), a programming task (max score 40 points)
Exam duration	Midterm exams: 90 minutes; exam 180 minutes	100 minutes	100 minutes	120 minutes	120 minutes

*ECTS – European Credit Transfer and Accumulation System

3.3. Sample

This subsection describes the samples used to obtain answers to the research questions. For RQ1, samples from the CP and IPM were used. Three samples were used from the CP course (participants of the first midterm exam, the second midterm exam, and both midterm exams), which included all students whose submitted logs were complete and did not contain pre-exam activities that were not related to the exam (for more details, see Table 3). One sample was used from the IPM course. First, every tenth log of those students' logs was selected who passed the exam, and 90 logs were collected in this stage. Then, each log was analyzed, and those that were correct, contained information only about solving the exam task, and were in one file, were selected.

Table 3. Background information for RQ1

Course	**CP**			**IPM**
	First midterm exam	**Second midterm exam**	**Participated in both midterm exams**	**Exam**
N	301	275	233	27
Experience	Beginners 79 (26.2%) Non-beginners 222 (73.8%)	Beginners 72 (26.2%) Non-beginners 203 (73.8%)	Beginners 61 (26.2%) Non-beginners 172 (73.8%)	-
Gender	Men 179 (59.5%) Women 122 (40.5%)	Men 165 (60%) Women 110 (40%)	Men 138 (59.2%) Women 95 (40.8%)	Men 15 (56%) Women 12 (44 %)

For RQ2, data from two years (2020 and 2021) from the course IP1 were used. An experimental group and a control group were formed for the experiment. Those students who participated in the exam and answered the questionnaire about previous programming experience were included in the groups (57 students in 2020 and 68 students in 2021). The groups were then balanced based on the following characteristics: previous programming experience, gender, and mandatory nature of the course. Finally, the Chi-Square test was used to check for group balance. More information about the groups can be found in Table 4.

For RQ3, one sample from the courses IP1 and IP2 was used. The sample consists of students who in 2020 studied in courses IP1 and IP2, submitted exam solutions and log files for both exams, and submitted log files of homework tasks for at least 50% of the weeks. Seventeen students met those conditions. In addition, the logs and submitted solutions from the two courses were used to validate the plagiarism detection tool Thonny Log Analyzer. One of these log and solution

Table 4. Background information for RQ2

Course	**IP1**	
Group	**Experimental group**	**Control group**
N	25	55
Experience	Beginners 13 (52%) Non-beginners 12 (48%)	Beginners 28 (51%) Non-beginners 27 (49%)
Gender	Men 8 (32%) Women 17 (68%)	Men 17 (31%) Women 38 (69%)

collections includes data from the course IP1 in 2022, and the other from the TCC course in 2023. Additional information about the samples for RQ3 is presented in Table 5.

Table 5. Background information for RQ3

Course	**IP1 and IP2**	**TCC**	**IP1**
N	17	141	44
Gender	Men 5 (29.4%) Women 12 (70.6%)	Men 76 (53.9%) Women 65 (46.1%)	Men 8 (18.2%) Women 36 (81.8%)

3.4. Data Collection

This subsection describes the data collection process. It is important to note that informed consent was obtained from participants for the studies for this thesis. We likewise have permission from the Research Ethics Committee of the University of Tartu (341/T-2, 2021–2026). For all research questions, information retrieval from Thonny logs was used to obtain behavior features in programming. Submission of logs was mandatory for all courses. Students submitted Thonny logs in Moodle. Selected information about the programming process was extracted from each student's logs into a CSV file. For RQ1, the log data from IPM was additionally collected using Thonny's functionality of replaying the programming process. In addition to the log data, information was obtained from pre-questionnaire about previous programming experience, from Moodle learning management system and the Study Information System (SIS). Studies (O'Malley & Aggarwal, 2020; Veerasamy et al., 2018; Vihavainen et al., 2014b) have found that prior experience in coding affects performance in introductory programming courses. Therefore, it was considered as well. A survey was conducted at the start of the course to get information on whether students had prior experience. This study employed a single multiple-choice question that provided the options presented in Table 6. For this study, the results were categorized based on programming experience into two groups: non-beginners (students who selected options 2, 3, or 4 on the question, indicating prior programming experience) and beginners (students who selected

option 1, "I have never attempted programming"). More information about data collection for RQ1 is given in Table 6.

Table 6. Data collection for RQ1

Course	CP	IPM
Data from Thonny logs	Task start time Task end time Number of runs Number of error messages Number of error messages by type Number of pastes Number of pasted characters Number of debugs Number of files opened Time solving till first run Number of characters at first run for each program Number of characters at log submission for each program	Task start time Task end time Error messages with timestamps The time between an error message and next run Runs with timestamps Runs with and without error messages Use of debugger with a timestamp Number of previous solutions opened Parts of program written before the first execution
Pre-questionnaire	About previous programming experience Single select multiple choice question answers: 1) I have never attempted programming, 2) I have experimented with programming but made minimal progress, 3) I am capable of creating basic programs, 4) I possess strong programming skills, and this course offers me little new knowledge.	-
Data from Moodle	Scores for programming tasks in two midterm exams (max score 20 points for each)	Pass/fail exam
Data from SIS	Gender	Gender

For RQ2, the main research method was an experiment during which data were collected. Thonny logs were used to get information about features in programming (see Table 7) and more detailed information to give log-based feedback. The lecturer provided written feedback to the experimental group on the programming

process in Moodle after the homework deadlines, in addition to the comments on errors and style that were given to all groups. Furthermore, all groups received automated instant feedback before the homework deadline, which consisted of elaborated feedback according to Narciss's (2008) classification, more precisely, knowledge about mistakes (e.g., test failures with explanations, types of error messages with clarifications, error locations). Students were allowed to resubmit their homework multiple times until the automated assessment reached a "passed" status. The log-based feedback can be categorized as knowledge about how to proceed, as it concerns learners' problem-solving strategies and improving programming techniques useful to proceed with tasks, as well as knowledge about meta-cognition, because it helps them improve their general programming techniques and evaluation of their work and strategies (Keuning et al., 2018; Narciss, 2008; Narciss, 2013).

To give feedback based on the logs, the lecturer examined the homework logs of all students in the experimental group each week, utilizing Thonny's feature of replaying the programming process. This Thonny's feature provides character-level playback of students' programming activity from start to finish. It enables the observer to see the exact sequence in which the program was constructed, when parts were removed, when the program was executed, what the output was, and how the student responded to any error messages. Therefore, the lecturer was able to provide feedback on the programming process itself. If the logs revealed inefficient programming techniques, recommendations were provided for improvement. For instance, if the log indicated that the student was attempting to resolve an error message primarily through trial and error, the lecturer recommended, in individual feedback, using the Thonny debugger. Here is one example: "Perhaps stepping through both your own solutions and the while-loop examples from the material with a debugger (click the bug icon next to the run button, then keep pressing F7) will help you see how the program works". Similarly, if the student had not tested a function independently but instead wrote the entire program at once and subsequently struggled to find errors, the lecturer suggested testing the function separately in individual feedback. When effective working practices were observed, these were also acknowledged in the feedback. General recommendations were also provided during the practical session based on the information obtained from the logs. For example, if the logs revealed that many students struggled to understand loops while working on their homework, the concept was revisited in greater detail during the practical session. In particular, it was demonstrated how to use the print function to trace the sequence of commands.

In addition, a pre-questionnaire was conducted to get information about students' previous programming experience. This study used a single multiple-choice question, with the response options presented in Table 7. Participants were grouped by programming experience into two categories: non-beginners (those who chose options 2 or 3, indicating some prior programming experience) and

beginners (those who selected option 1, "I have never attempted programming"). For exam scores and students' background information, the learning management system Moodle and the Study Information System were used. More details about data collection for Q2 are available in Table 7.

Table 7. Data collection for RQ2

Courses	IP1 and IP2
Data from Thonny logs	Solution time Number of runs Number of error messages Copying-pasting
	Programming-process information for log-based feedback (using Thonny's functionality of replaying the programming process)
Pre-questionnaire	About previous programming experience Single select multiple choice question answers: 1) I have never attempted programming, 2) I have experimented with programming but made minimal progress, or 3) I possess strong programming skills, and this course offers me little new knowledge.
Data from Moodle	Scores on the exam tasks: Test questions (max score 24 points) A programming task (max score 24 points)
Data from SIS	Gender Specialty Field of study Mandatory nature of the course

For RQ3, Thonny logs were used to obtain information about general style features and programming style features that can be used to detect plagiarism. The plagiarism detection tool Thonny Log Analyzer was used to control whether information from Thonny logs — including determining the number of executions, total time spent coding, file size, and the ratio of pasted text to manually typed text, identifying duplicate submissions, detecting identical pasted texts, recognizing the same source code pasted in different log files, and assessing source code similarity — can be used to check for plagiarism.

3.5. Data Analysis

This subsection gives an overview of the data analysis process. The main approach is quantitative. A quantitative approach was selected as it relies on objective measurements, enabling the identification of patterns and trends through data

and extrapolating findings to a larger population (Dehalwar & Sharma, 2024). The data were processed and examined using IBM SPSS Statistics software.

For RQ1, cluster analysis was used for forming one classification, and the other one, with a small sample, was done manually based on the most obvious differences in behavior features in programming, applying a top-down strategy. The behavior features in programming underlying both analyses are presented in Table 8. First, the cluster analysis is presented and the manual grouping is described after that. Clustering is a useful technique for uncovering hidden patterns in datasets, and previous studies frequently employ the k-means algorithm to identify solver types using programming-process data (e.g., Bey et al., 2019; Pereira et al., 2020; Shrestha et al., 2022; Zhang et al., 2023b). Samples from CP were used for the cluster analysis (see Table 3). The four features in programming, such as the number of runs, the number of error messages, the percentage of typed characters at the first execution, and the percentage of syntax errors, were applied in the analysis. These were mainly selected based on previous research. However, one behavior feature in programming is new – previous studies have not incorporated the percentage of typed characters at the first execution. Furthermore, the cluster analysis testing phase incorporated additional features, including task completion time and debugging instances, but these were not as meaningful. The percentage of characters typed before the first run is a crucial feature in programming, which has not been considered in previous research analyzing exam score differences among various solver types, both beginners and non-beginners. This metric could be significant, as it reflects a delayed start to debugging, potentially correlating with exam scores. The number of runs and error messages are included because a high count may indicate many trial-and-error attempts, suggesting ineffective behavior (Jemmali et al., 2020; Sharma et al., 2018). At the same time, the relatively high number of errors may also be caused by frequent testing, and it can be indicative of inefficient behavior if students avoid debugging at all and do not execute their programs (Carter & Hundhausen, 2017). Including features in programming related to potentially ineffective behavior (such as a high number of error messages and a sizeable percentage of syntax errors) also aids in identifying which other features are typically associated with these behaviors.

The next step involved standardizing the feature scores to a consistent scale using the z-score. Each dataset was analyzed using k-means cluster analysis, with a maximum of 10 iterations. The goal was to categorize students based on features in programming. Since k-means cluster analysis often requires testing different cluster numbers to identify the most meaningful solution for the research context (Jain, 2010), we experimented with cluster models containing three, four, five, six, and seven clusters. The analysis also included various programming features in addition to the final ones. The results indicated that the most meaningful solution was achieved with four clusters. The overview of the analysis methods is presented in Table 9. An Alluvial diagram was employed to visualize the transitions between clusters.

The sample from IPM (see Table 3) was used to create an alternative set of solver types. Clear distinctions were observed in solving time, frequency of error messages, reliance on previous solutions, etc., between learners who approached the task gradually and those who did not. As a result, learners were initially categorized based on whether they followed an incremental solving process. Further analysis was then conducted to identify differences within these groups, which led to dividing learners into three distinct categories based on these additional variations.

For RQ2, the experimental and control groups' exam test scores and exam programming task scores were first compared, with whole groups, beginners, and non-beginners analyzed separately. Additionally, the groups were compared based on the behavior features in programming outlined in Table 8. The features in programming for the analysis were chosen based on the results of previous research. The analysis methods and their details are presented in Table 1.

For RQ3, the logs were examined to identify characteristics that distinguish students' programming styles. All logs from the two courses included in the sample were analyzed using Thonny's programming-process replaying functionality. Additionally, the submitted programs were reviewed. Before the analysis, a table listing characteristics that could differentiate students was created. Each week, information about individual students was added to the table during the log analysis. The data collected from each student's logs over different weeks was then compared. Furthermore, the degree to which students' programming style features aligned with those found in the study materials was also assessed.

Table 8. The main behavior features and references for RQ1 and RQ2

Feature	Explanation	References	RQ
Number of runs	The total count of the program executions during the solution(s) of exam programming task(s)	Blikstein, 2011; Jadud, 2006; Sharma et al., 2018; Tabanao et al., 2011; Vihavainen et al., 2014a	RQ1, RQ2
Number of error messages	The total count of error messages encountered during the solution of two midterm exam programming tasks	Blikstein, 2011; Carter et al., 2015; Jadud, 2006; Price et al., 2020; Sharma et al., 2018; Tabanao et al., 2011; Vihavainen et al., 2014a; Watson et al., 2013	RQ1, RQ2
Percentage of typed characters at the first execution	The proportion of characters typed before the first execution in two midterm exam programming tasks	The percentage of typed characters before the first run has not been incorporated in previous works. In other prior studies, the amount of code written between executions or submissions (Bey & Champagnat, 2022; Pereira et al., 2020; Zhang et al., 2023b), as well as execution-editing sequences (Carter & Hundhausen, 2017), have been taken into account.	RQ1
Percentage of syntax errors	The percentage of syntax errors relative to the total number of errors encountered while solving two midterm exam programming tasks	Bey et al., 2019; Estey & Coady, 2016; Pereira et al., 2019; Pereira et al., 2020; Pereira et al., 2021	RQ1
Completion time	Completion time of exam programming task	Matsuzawa et al., 2013; Pereira et al., 2019; Vihavainen et al., 2014b	RQ1, RQ2
Number of files opened	Number of files opened during solving	Blikstein, 2011	RQ1
Number of pastes	The total count of pastes during solving of the exam programming task	Blikstein, 2011; Vihavainen et al., 2014a	RQ2

4. RESULTS

This chapter presents the key findings based on the three research questions addressed in the doctoral thesis. Subchapter 4.1 describes solver types, their differences in midterm exam scores, and the persistence of types. Subchapter 4.2 deals with log-based feedback and how it affects exam scores, task-solving time, number of runs, error messages, and pastes. Subchapter 4.3 presents behavior features in programming that can be used to detect plagiarism.

4.1. Analyzing Solver Types: Forming, Variations and Persistence

This subchapter addresses the first research question, with more detailed results available in Publications I, IV, and VI.

4.1.1. Solver Types based on Programming-Process Analysis

The following solver types were identified using cluster analysis: (1) "Frequent pressers of the run button", (2) "Receivers of syntax errors", (3) "Balanced solvers", and (4) "Late starters of the program execution". Among both beginners and non-beginners, clusters that shared characteristics with the overall group clusters were identified. Figures 2 and 3 present the whole group clusters for the first and second midterm exams, and Figure 4 presents the beginner and non-beginner clusters for the first and second midterm exams.

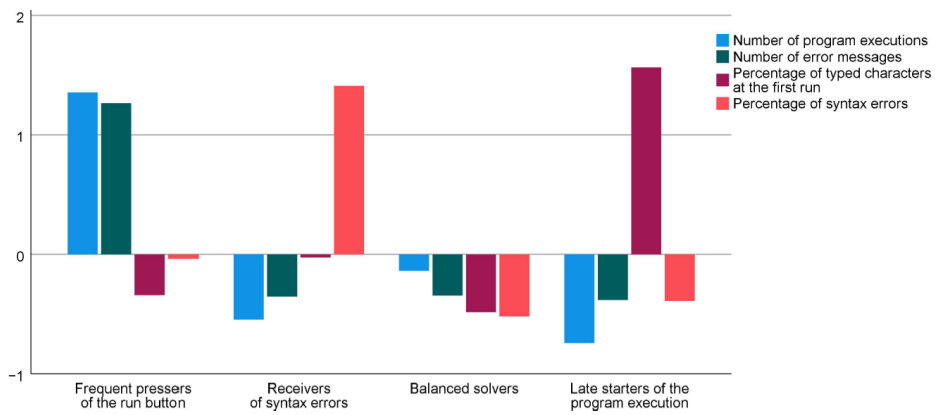

Figure 2. Clusters of the whole group at midterm exam 1

The groups can be characterized as follows: "**Frequent pressers of the run button**" stand out by executing programs more often and receiving more error messages than others (in all cases, $p < 0.001$). "**Receivers of syntax errors**" are characterized by the highest proportion of syntax errors (in all cases, $p <$

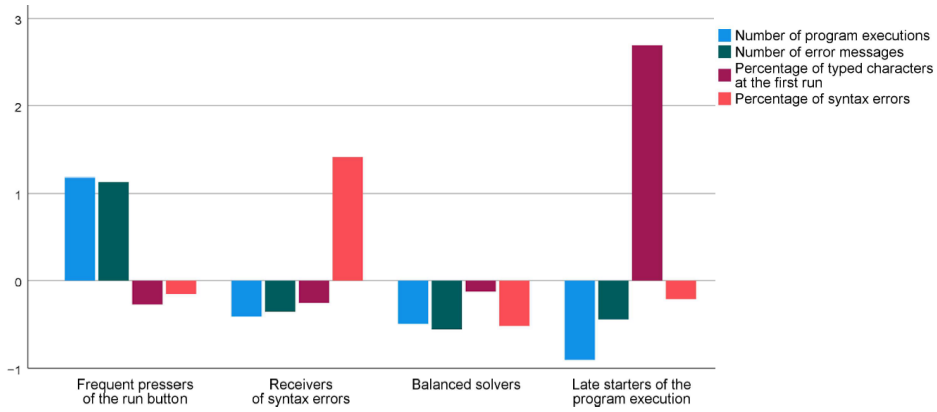

Figure 3. Clusters of the whole group at midterm exam 2

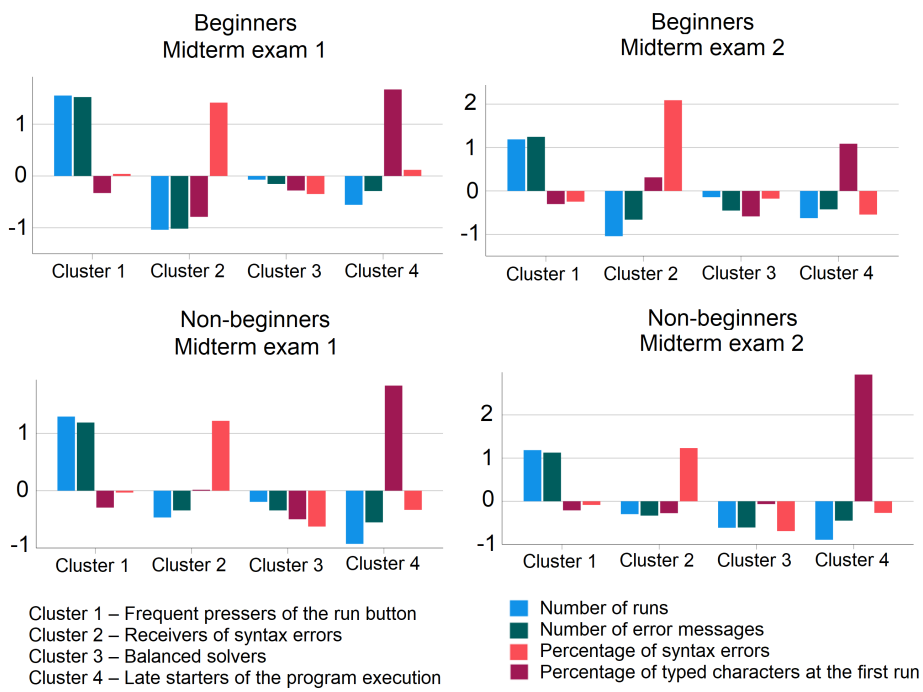

Figure 4. Clusters of beginners and non-beginners at midterm exams 1 and 2

0.05). "**Balanced solvers**" exhibit below-average values across all features. Finally, "Late starters of program execution" had the highest proportion of written program characters before their first execution (in all cases, $p < 0.001$).

Using a different method and a smaller sample, another set of groups was identified and labeled as “Bricklayers”, “Stonecutters”, and “Masters”. “**Bricklayers**” approached the task step by step, similar to how a bricklayer builds a wall gradually. They ran the program for the first time after writing a function. Throughout the programming process, they corrected errors and needed little to no use of previous solutions while receiving a fair number of error messages. “**Stonecutters**” tested their work for the first time only after completing most of the solution. This can be likened to a stonecutter’s work, which requires raw material before starting to shape the object. It is a labor-intensive process that demands considerable effort and various tools. “Stonecutters” needed a longer time and received more error messages than any Bricklayer. All Stonecutters received repeated error messages and needed 5-21 previous solutions. “**Masters**” tested their work for the first time after writing most of the solution, but their code was already nearly correct. As a result, they received fewer error messages and encountered fewer difficulties than “Stonecutters”. Examples of error messages received during the programming task-solving process of a Bricklayer, a Stonecutter, and a Master are presented in Figure 5.

4.1.2. Variations in Midterm Exam Scores of Solver Types

To analyze midterm exam scores, the scores of all clusters were compared with each other. Table 9 presents the descriptive statistics, while the boxplots in Figures 6 and 7 illustrate the distribution of midterm exam scores. Regarding the first midterm exam, the Mann-Whitney U test revealed a statistically significant difference between "Balanced solvers" and "Receivers of syntax errors" as well as between "Balanced solvers" and "Late starters of the program execution." Specifically, the midterm exam scores of "Balanced solvers" were higher than those of "Receivers of syntax errors" ($U = 2835$, $p < 0.05$) and "Late starters of the program execution" ($U = 2493.5$, $p < 0.05$). For the second midterm exam, "Balanced solvers" got higher exam scores than "Frequent pressers of the run button" ($U = 2503$; $p < 0.05$) and "Late starters of the program execution" ($U = 539$; $p < 0.05$). In addition, "Receivers of syntax errors" performed better than "Frequent pressers of the run button" ($U = 1434$; $p < 0.05$) and "Late starters of the program execution" ($U = 296$; $p = 0.05$).

Additionally, midterm exam scores were compared between clusters by separately analyzing the scores of beginners and non-beginners on both midterm exams. The descriptive statistics are provided in Table 10, while Figure 8 illustrates the distribution of midterm exam scores for beginners and non-beginners by clusters. The Mann-Whitney U test revealed that when beginners and non-beginners were analyzed separately, there were no statistically significant differences in their

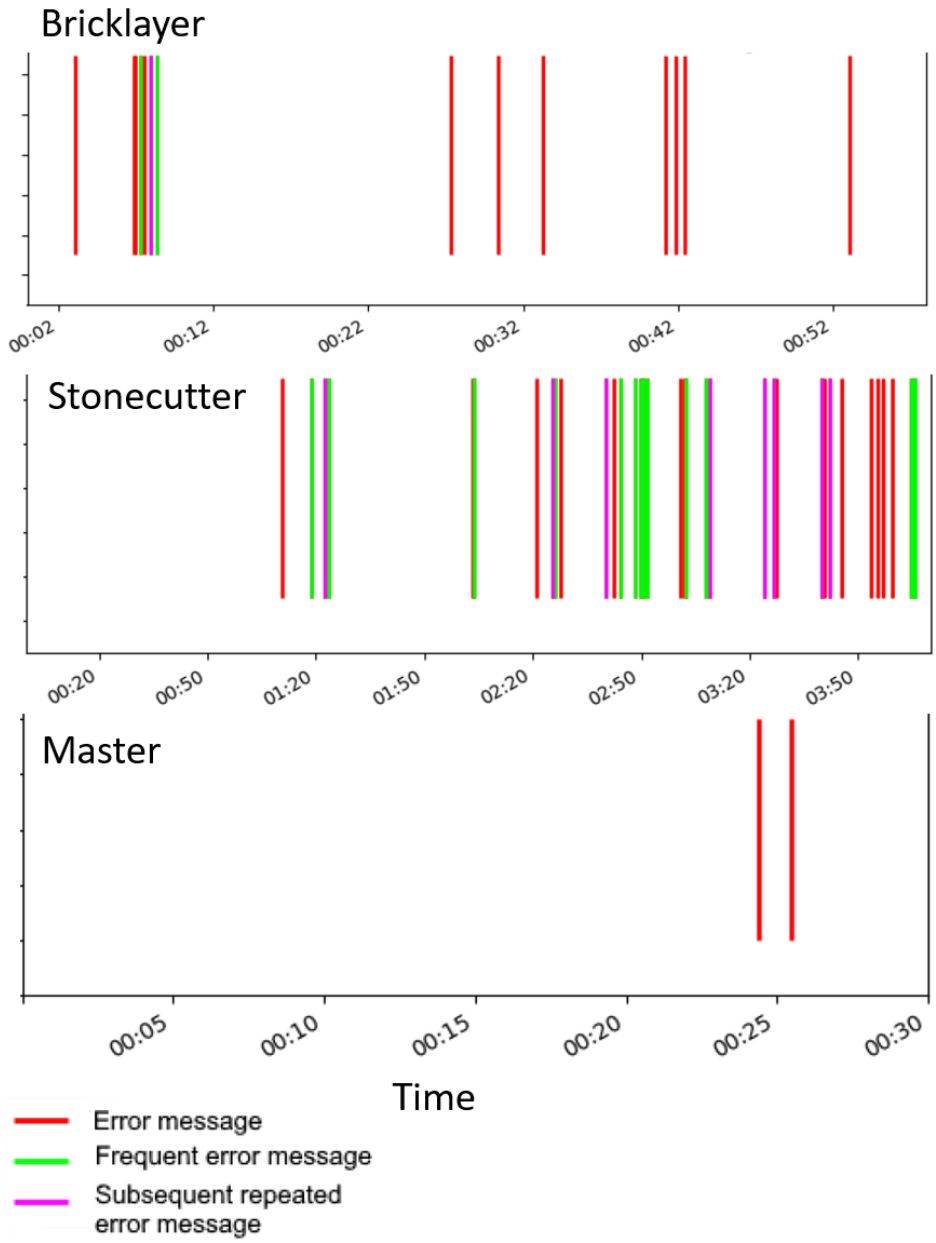

Figure 5. Examples of error messages of a Bricklayer, a Stonecutter, and a Master

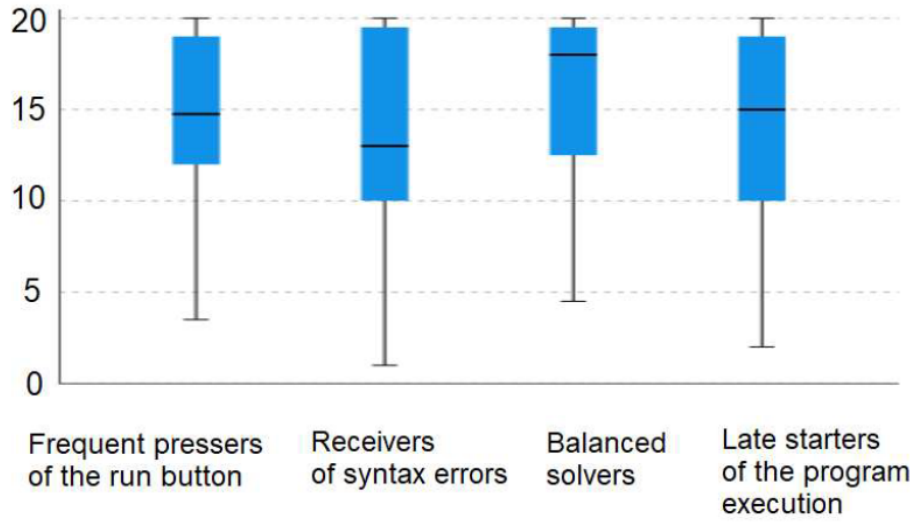

Figure 6. Distribution of midterm exam scores for the whole group at midterm exam 1 by clusters

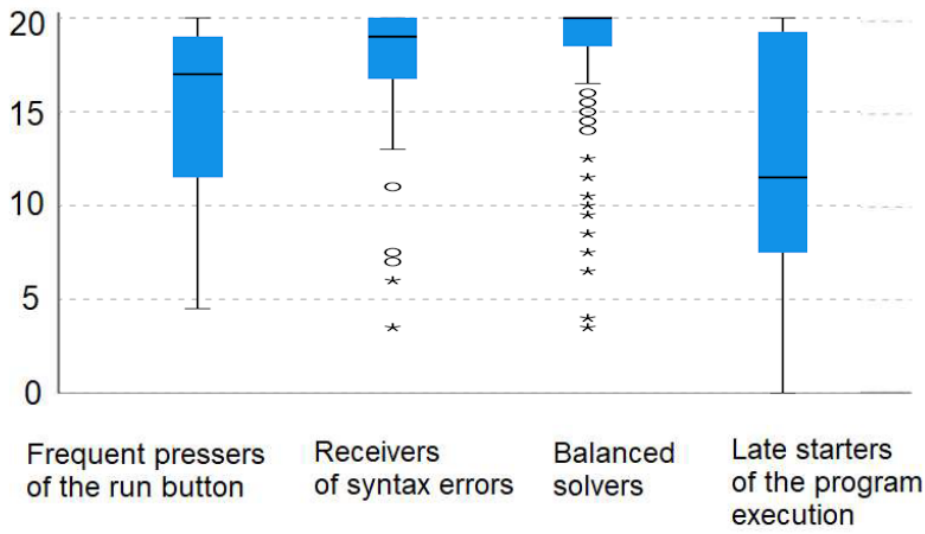

Figure 7. Distribution of midterm exam scores for the whole group at midterm exam 2 by clusters

Table 9. Descriptive statistics of both midterm exam scores for the whole group by clusters

Cluster	Exam	Students	Mean	SD	Min	Max
Frequent pressers of the run button	1	66 (21.9%)	14.85	4.28	3.5	20
	2	83 (30.2%)	15.19	4.37	4.5	20
Receivers of syntax errors	1	61 (20.3%)	13.63	5.06	1	20
	2	55 (20%)	17.20	4.10	3.5	20
Balanced solvers	1	121 (40.2%)	15.92	4.37	4.5	20
	2	118 (42.9%)	17.99	3.72	3.5	20
Late starters of the program execution	1	53 (17.6%)	13.85	5.51	2	20
	2	19 (6.9%)	11.87	7.02	0	20

first midterm exam scores across clusters. However, for the second midterm exam, statistically significant differences were observed among clusters for both beginners and non-beginners. Among beginners, those in the "Balanced solvers" cluster had significantly higher second midterm exam scores than those in the "Receivers of syntax errors" ($U = 58$, $p < 0.05$) and "Late starters of the program execution" ($U = 129$, $p < 0.05$) clusters. For non-beginners, "Balanced solvers" outperformed "Frequent pressers of the run button" ($U = 1058$, $p < 0.001$) and "Late starters of the program execution" ($U = 251$, $p < 0.05$). Additionally, "Receivers of syntax errors" scored significantly higher than "Frequent pressers of the run button" ($U = 698$, $p < 0.001$) and "Late starters of the program execution" ($U = 165$, $p < 0.05$).

Table 10. Descriptive statistics of midterm exam scores for beginners and non-beginners

	Cluster	Exam	Students	Mean	SD	Min	Max
Beginners	Frequent pressers of the run button	1	13 (16.5%)	12.04	3.74	7.5	19
		2	20 (27.8%)	14.68	4.64	7	20
	Receivers of syntax errors	1	9 (11.4%)	10.61	6.85	6	20
		2	9 (12.5%)	10.61	6.84	1	20
	Balanced solvers	1	43 (54.4%)	12.71	4.54	5	20
		2	26 (36.1%)	16.83	4.19	6.5	20
	Late starters of the program execution	1	14 (17.7%)	9.93	5.05	3.5	19.5
		2	17 (23.6%)	11.82	6.09	3.5	20
Non-beginners	Frequent pressers of the run button	1	54 (24.3%)	15.79	4.11	3.5	20
		2	62 (30.5%)	15.44	4.17	4.5	20
	Receivers of syntax errors	1	53 (23.9%)	14.89	3.6	3.5	20
		2	51 (25.1%)	18.51	3.24	3.5	20
	Balanced solvers	1	84 (37.8%)	16.55	4.03	4.5	20
		2	79 (38.9%)	16.83	2.77	6.5	20
	Late starters of the program execution	1	31 (14%)	16.35	4.29	2	20
		2	11 (5.4%)	14	7.07	0	20

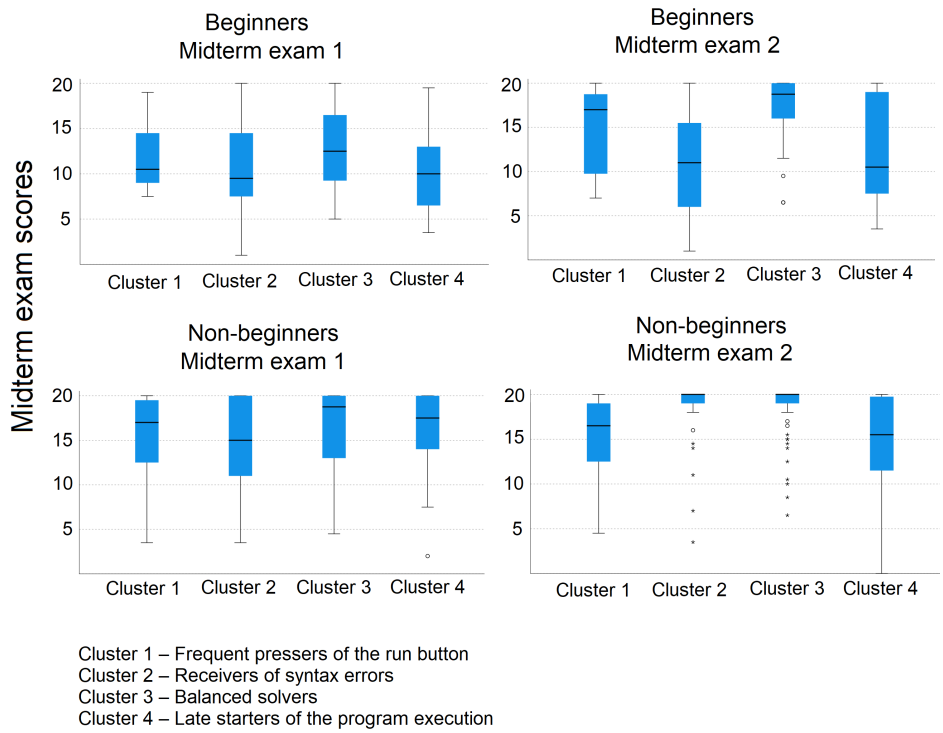

Figure 8. Distribution of midterm exam scores for beginners and non-beginners by clusters

4.1.3. Persistence of Solver Types Over Time

Two Alluvial diagrams illustrate the transitions of beginners and non-beginners between clusters (see Figures 9 and 10). Among the 61 beginners, 37 (61%) moved to different clusters, while 24 (39%) remained in the same cluster. Table 11 provides further details on the movement of beginners between clusters.

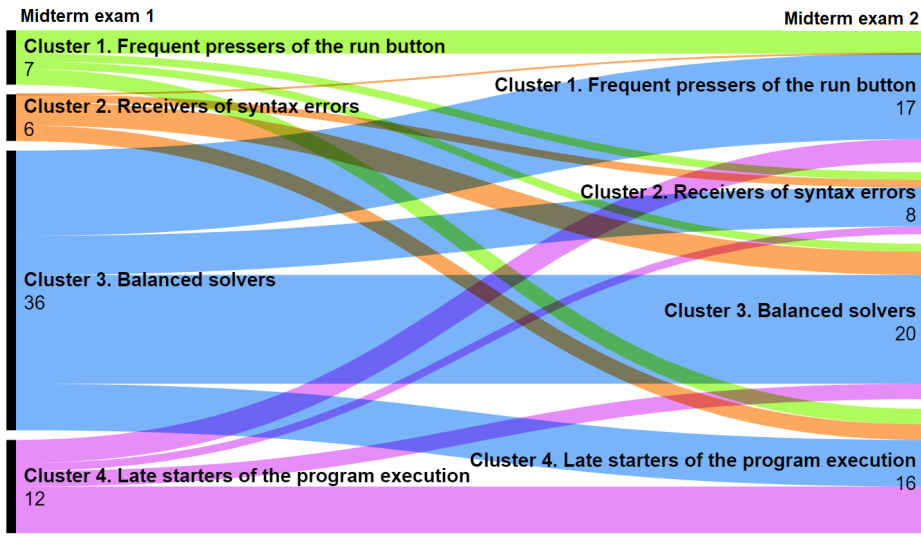

Figure 9. Movement of beginners between clusters

Table 11. Movement of beginners between clusters from midterm exam 1 (rows) to midterm exam 2 (columns)

	Cluster 1 (n = 17)	**Cluster 2** (n = 8)	**Cluster 3** (n = 20)	**Cluster 4** (n = 16)
Cluster 1. Frequent pressers of the run button (n = 7)	3 (43%)	1 (14%)	1 (14%)	**2 (29%)**
Cluster 2. Receivers of syntax errors (n = 6)	0 (0%)	1 (17%)	**3 (50%)**	2 (33%)
Cluster 3. Balanced solvers (n = 36)	**11 (31%)**	5 (14%)	14 (39%)	6 (17%)
Cluster 4. Late starters of the program execution (n = 12)	**3 (25%)**	1 (8%)	2 (17%)	6 (50%)

*A gray background indicates remaining in the same cluster, while **bold numbers** represent the highest transition probability.*

The proportion of transitions among non-beginners is similar to that of beginners. Figure 10 displays the movement of non-beginners between clusters. Of 172 non-beginners, 101 (59%) transitioned to different clusters while 71 (41%) remained the same. Table 12 provides more details on the movement of non-beginners between clusters.

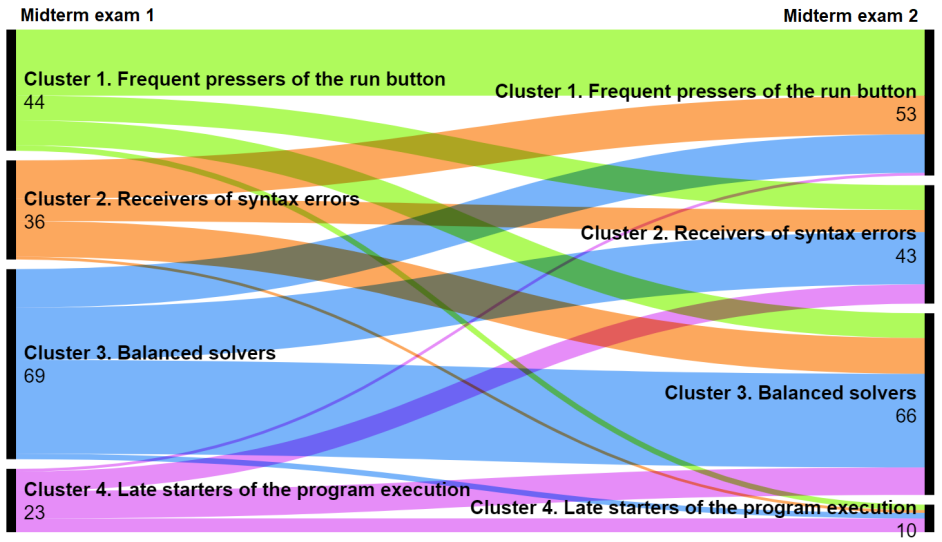

Figure 10. Movement of non-beginners between clusters

Table 12. Movement of non-beginners between clusters from midterm exam 1 (rows) to midterm exam 2 (columns)

	Cluster 1 (n = 53)	**Cluster 2** (n = 43)	**Cluster 3** (n = 66)	**Cluster 4** (n = 10)
Cluster 1. Frequent pressers of the run button (n = 44)	24 (55%)	**9 (20%)**	**9 (20%)**	2 (5%)
Cluster 2. Receivers of syntax errors (n = 36)	**14 (39%)**	8 (22%)	13 (36%)	1 (3%)
Cluster 3. Balanced solvers (n = 69)	14 (20%)	**19 (28%)**	34 (49%)	2 (3%)
Cluster 4. Late starters of the program execution (n = 23)	1 (4%)	7 (30%)	**10 (43%)**	5 (22%)

*A gray background indicates remaining in the same cluster, while **bold numbers** represent the highest transition probability.*

4.2. Influence of Log-Based Feedback on Students' Performance

This subchapter addresses the second research question, with more detailed results available in Publication III.

4.2.1. Influence on Exam Scores

First, the exam test scores for the experimental and control groups were compared (see Table 13). The Mann-Whitney U test indicated that the experimental group

had significantly higher exam test scores ($U = 432.5$, $p < 0.05$). While the experimental group also outperformed the control group in the programming task, the difference was not statistically significant ($U = 648.5$, $p > 0.05$).

Table 13. Summary statistics of exam test and programming task scores

	Group	N	Min	Max	M	SD	Mdn	Sk	K
Test scores	Exp	25	16	24	21.7	2.4	22	-0.735	-0.339
	Control	55	10.8	24	19.8	3.2	20	-0.841	0.583
Task scores	Exp	25	14	24	22.4	2.6	23.5	-2.045	3.865
	Control	55	10	24	21.1	4.1	23.5	-1.395	0.813

Exp - Experimental; N - Number; Min - Minimum; Max - Maximum; M - Mean; SD - Standard Deviation; Mdn - Median; Sk - Skewness; K - Kurtosis.

A separate analysis of beginners and non-beginners was also conducted (see Figures 11 and 12). The Mann-Whitney U test revealed that beginners in the experimental group achieved higher exam test scores, with the difference being statistically significant ($U = 78$, $p < 0.05$). Their programming task scores were also higher, though the difference was not statistically significant ($U = 170$, $p > 0.05$). On average, beginners in the experimental group scored 22.0 points on the exam test and 21.5 on the programming task, whereas beginners in the control group scored 19.6 and 20.5 points, respectively. For non-beginners, those in the experimental group outperformed those in the control group in both the exam test and programming task, but these differences were not statistically significant (Test: $U = 130.5$, $p > 0.05$; Programming Task: $U = 154.5$, $p > 0.05$). On average, non-beginners in the experimental group scored 21.4 points on the exam test and 23.3 on the programming task, while those in the control group scored 20.0 and 21.8 points, respectively.

4.2.2. Influence on Programming Task Solving Time, Number of Runs, Error Messages, and Pastes

To provide a more detailed description of the programming task solution beyond exam scores, solution time, the number of runs, error messages, and pastes were analyzed. The summary statistics for these indicators are presented in Table 14.

The Mann-Whitney U test indicated that the experimental group completed the programming task more quickly than the control group, with the difference being statistically significant ($U = 458.5$, $p < 0.05$). The density plot (see Figure 13) illustrates the distribution of task completion times.

A separate analysis of beginners and non-beginners was also conducted. The Mann-Whitney U test showed that beginners in the experimental group completed the programming task significantly faster ($U = 100.5$, $p < 0.05$). The average task-solving time for experimental group beginners was 53 minutes, while control group beginners took 72.1 minutes. Non-beginners in the experimental group

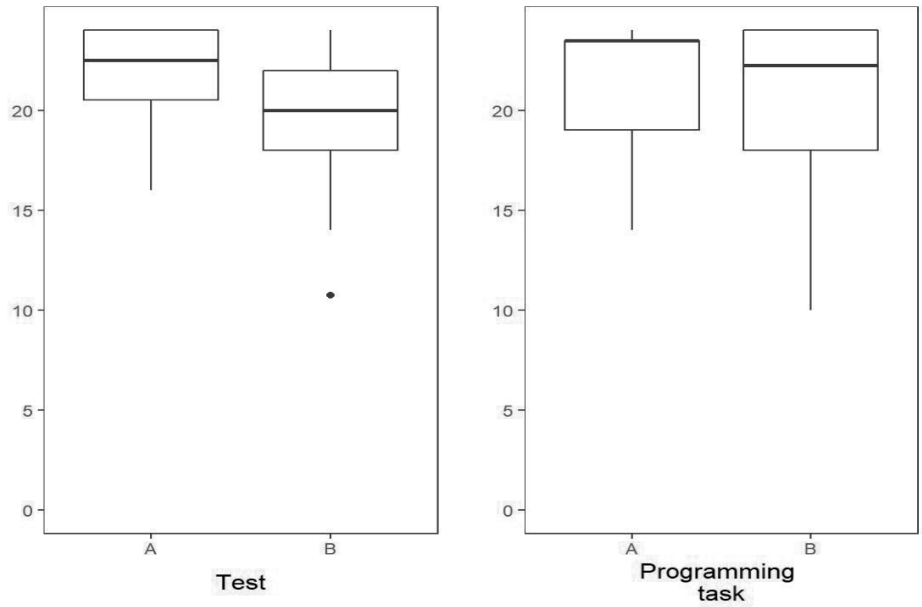

Figure 11. Exam test and programming task scores of beginners (A - experimental group, B - control group)

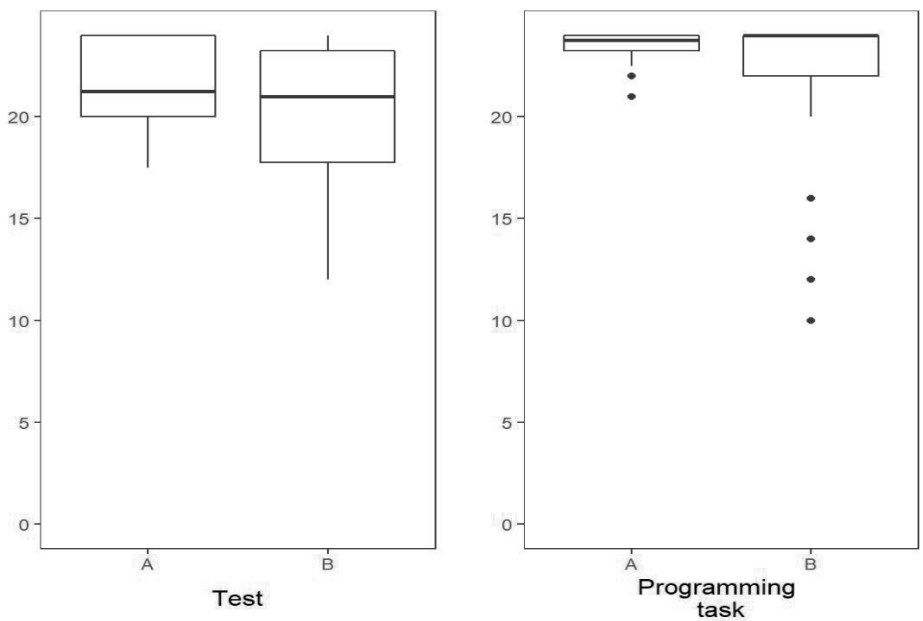

Figure 12. Exam test and programming task scores of non-beginners (A - experimental group, B - control group)

Table 14. Summary statistics of programming task solution time, the number of runs, error messages, and pastes

	Group	N	Min	Max	M	SD	Mdn	Sk	K
Solution time	Exp	25	22	95	49.1	21.3	46	0.378	0.922
	Control	55	22	96	63	24.5	66	0.267	-1.322
N of runs	Exp	25	2	45	22.2	13.6	19	0.366	-1.150
	Control	55	2	83	27.4	21	19	1.220	0.971
N of error messages	Exp	25	0	31	8.8	8.5	6	1.374	1.014
	Control	55	0	65	13	12.4	10	1.905	5.233
N of pastes	Exp	25	0	28	6.2	7.3	4	1.994	3.567
	Control	55	0	34	6	7.5	4	2.504	6.535

Exp – Experimental; N - Number; Min - Minimum; Max - Maximum; M - Mean; SD - Standard Deviation; Mdn - Median; Sk - Skewness; K - Kurtosis.

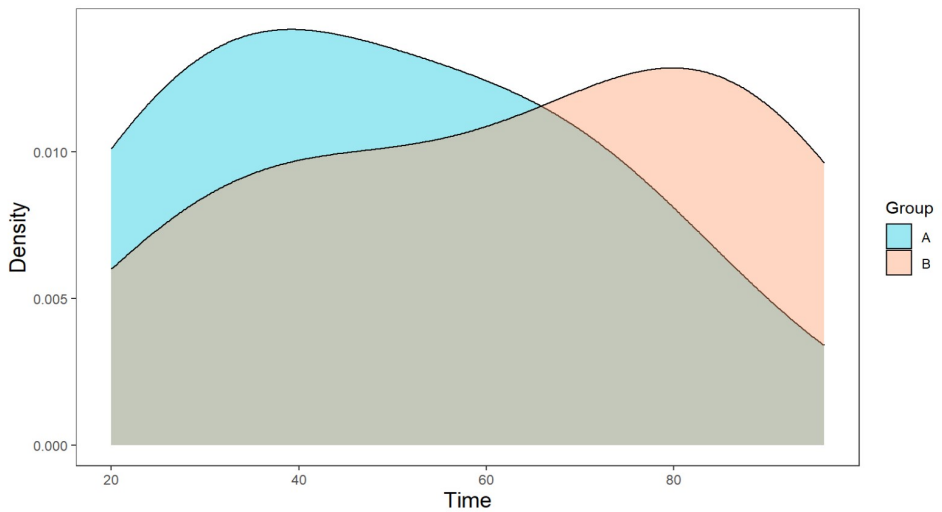

Figure 13. Time taken to solve the programming task (A - experimental group, B - control group)

completed the task faster than those in the control group, but the difference was not statistically significant ($U = 131$, $p > 0.05$). Non-beginners in the experimental group took an average of 44.8 minutes, compared to 53.6 minutes for non-beginners in the control group. The density plots (Figures 14 and 15) show the distribution of task-solving times for both beginners and non-beginners.

Students in the experimental group had fewer runs and error messages than the control group, although these differences were not statistically significant (Runs: $U = 625.5$, $p > 0.05$; Error messages: $U = 543.5$, $p > 0.05$). The control group had fewer pastes than the experimental group, but this difference was also insignificant ($U = 678$, $p > 0.05$). The Mann-Whitney U test revealed that beginners in the experimental group had a lower number of runs and error messages, but the difference was not statistically significant (Runs: $U = 155$, $p > 0.05$; Error messages: U

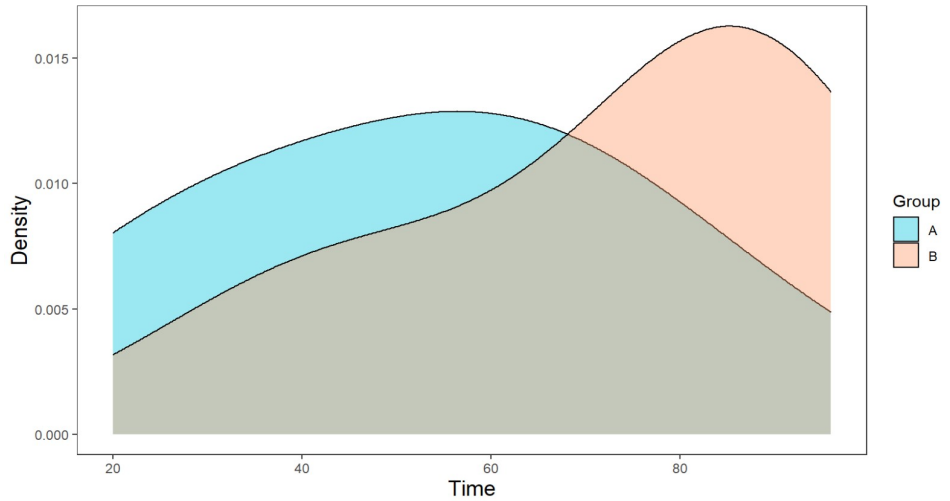

Figure 14. Beginners time taken to solve the programming task (A - experimental group, B - control group)

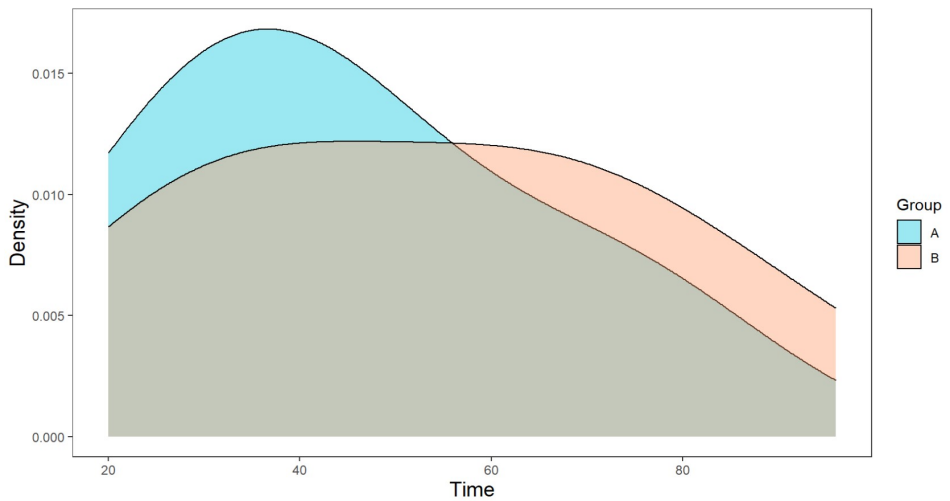

Figure 15. Non-beginners time taken to solve the programming task (A - experimental group, B - control group)

= 137.5, p > 0.05). Beginners in the experimental group had an average of 22.92 runs and 9.54 error messages during the programming task. In contrast, beginners in the control group had an average of 27.96 runs and 13.96 error messages. The control group beginners had fewer pastes (respectively 5.43 and 8.31), but this difference was also insignificant (U = 153, p > 0.05). Non-beginners in the experimental group also had a lower number of runs, error messages, and pastes compared to those in the control group, with no statistically significant differences (Runs: U = 148, p > 0.05; Error messages: U = 135.5, p > 0.05; Pastes: U = 138.5, p > 0.05). Non-beginners in the experimental group averaged 21.33 runs, 8 error messages, and 3.83 pastes in the programming task, while those in the control group averaged 26.89 runs, 12.04 error messages, and 6.59 pastes.

4.3. Behavior Features in Programming for Detecting Plagiarism

This subchapter addresses the third research question, with more detailed results available in Publications II, and V.

4.3.1. General and Programming Style Features

The study focused on two types of behavior features in programming related to style: (1) general style features and (2) programming style features. While programming style features refer to the variations related to the syntax options, general style features refer to general patterns that appear during the programming process, for example, the sequence in which brackets, quotation marks, apostrophes, and square brackets are written. For instance, some individuals may first write an opening bracket, followed by the text, and then the closing bracket (as shown in Table 15: (x) "x" [x]). Others may first write an opening bracket, immediately followed by the closing bracket, and then insert the text inside (as shown in Table 15: () "" []). Students were categorized into seven distinct types based on the primary sequence in which general style elements are written (Table 15).

Table 15. Using parentheses, quotation marks, apostrophes, and square brackets in writing

Type number	Type	Number of students
1	(x) “x” [x]	8
2	() "" []	4
3	(x) “x” []	1
4	() “x” []	1
5	(x) ’x’ []	1
6	Mixed: (x) “x” [x] and () "" []	1
7	Mixed: (x) ’x’ [x] and () "" []	1

Students' approach to writing brackets and other symbols remained consistent from week to week, showing little to no change throughout the course. The order in which they wrote these elements appeared intuitive, suggesting they did not consciously think about it. The study also examined whether students preferred quotation marks or apostrophes when programming. Of the 17 students, 15 primarily used quotation marks, as written in the study materials. Overall, their choice between quotation marks and apostrophes remained stable over time. Some students occasionally used apostrophes in specific contexts despite mainly using quotation marks. One student who typically used apostrophes, switched to quotation marks while working on the final assignment but later replaced them with apostrophes. This behavior could suggest potential plagiarism. The study also investigated whether students used spaces in their expressions. Among the 17 students, one wrote without spaces, two had an inconsistent style, and 14 primarily included spaces. Additionally, it was observed that the study materials influenced students' use of spaces, as spaces were generally present in the provided materials. The following part deals with style features more closely related to programming. One aspect examined was the choice between 'i += 1' and 'i = i + 1'. Students used both variations, often trying one before switching to the other. Toward the end of the course and during the second course, there was an increased use of 'i += 1' (Table 16). When comparing students' choices with the study materials, it became evident that the materials played a significant role in influencing their programming decisions.

Table 16. Using of `i = i + 1` and `i += 1` in programming

Introduction to programming			**Introduction to programming II**		
Week	**Style feature**	**N**	**Week**	**Style feature**	**N**
Week 3	only `i = i + 1`	1	Week 1	only `i = i + 1`	0
	only `i += 1`	9		only `i += 1`	14
	both	7		both	3
Week 4	only `i = i + 1`	3	Week 2	only `i = i + 1`	2
	only `i += 1`	13		only `i += 1`	11
	both	1		both	1
				missing	3
Week 6	only `i = i + 1`	1	–		
	only `i += 1`	16			
	both	1			
Final assignment	only `i = i + 1`	1	Final assignment	only `i = i + 1`	2
	only `i += 1`	15		only `i += 1`	15
	both	1		both	0

When opening a file, most students used the format 'file = open("data.txt", encoding="UTF-8")'. The alternative approach, 'with open("data.txt") as file', was not used by any students in week 4. However, by week 6, one student used it in a single task and continued using it consistently throughout the second course and the final assignment. The format 'file = open("data.txt", encoding="UTF-8")' was also the one presented in the study materials. Additionally, how students handled file opening was analyzed more thoroughly, including whether they specified encoding or included parameters like 'r'. The findings showed that their approach varied from week to week rather than remaining consistent. The results suggest that general style features tend to remain consistent over time. This consistency makes them useful for automatically verifying whether a student has completed a task independently. The use of programming style features undergoes more changes over time and is, therefore, unsuitable for plagiarism detection.

4.3.2. Plagiarism Detection Tool Using Behavior Features in Programming

Although using Thonny's functionality of replaying the programming process allows analyzing behavior features in programming related to style, it is time-consuming. So, using a tool for automatic programming-process-based plagiarism detection is reasonable. Thonny Log Analyzer was created to reduce the log analysis time and offer different analysis types based on the programming process. It is crucial that users can utilize a ZIP file containing all students' Thonny logs to conduct plagiarism analysis. Six types of plagiarism detection analysis, which are presented in Table 17, were used in Thonny Log Analyzer for creating the plagiarism detection functionality based on log data from the programming process. Users can also change specific settings according to their needs. For example, they can specify the proportion of pasted text at which the application should flag and display a corresponding case in a reviewable list. Figure 16 shows a view of one analysis type in the developed tool. To expand the plagiarism detection functionality to other languages, an IntelliJ Platform plugin was developed for generating logs. Additionally, a modified version of the Thonny Log Analyzer was developed to analyze these logs.

To evaluate the effectiveness of the new tool, it was tested on logs collected from various courses, which had been retrieved from Moodle. The same sets of student solutions were also examined using the VPL plug-in, and the outcomes were compared. The new tool detected instances of plagiarism that the other tool had failed to identify. One such case was found in the TCC course. During the exam, one student used ChatGPT, resulting in a program that was not similar to any other student's work. Consequently, the VPL program similarity analysis was unable to find it. However, the new tool raised an alert through two types of analysis. The student work analysis flagged the case due to the short total working time of 13 minutes, while the pasted text percentage analysis highlighted

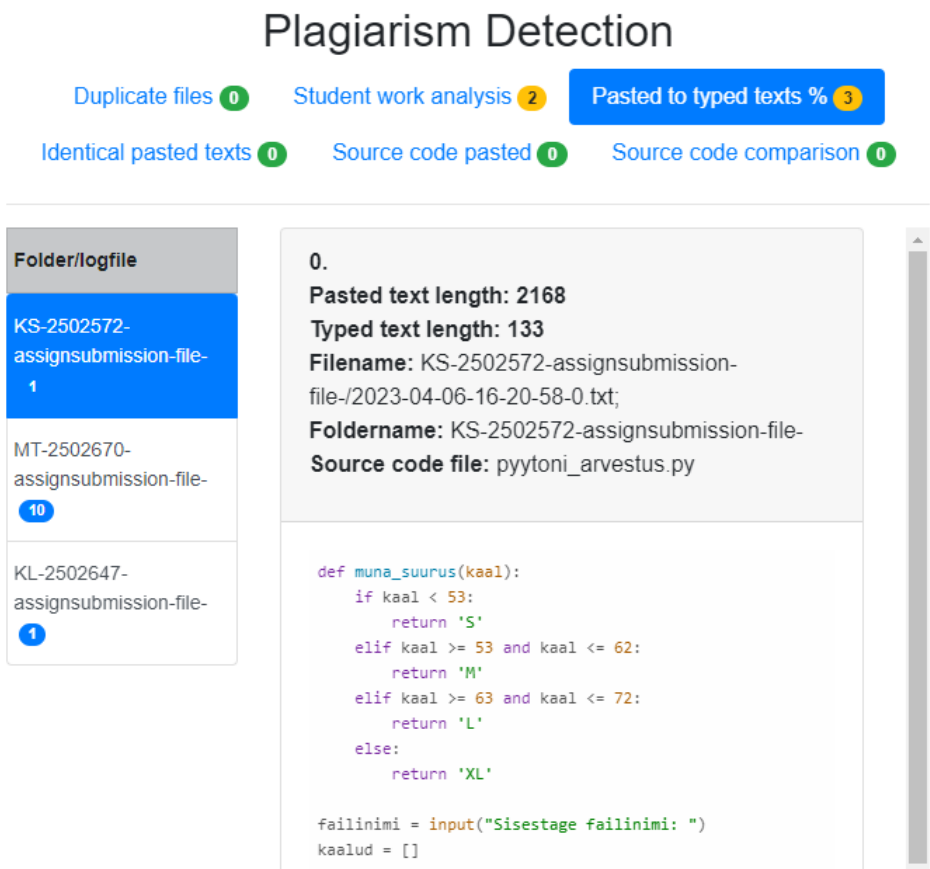

Figure 16. Pasted texts percentage analysis

Table 17. Types of plagiarism detection analysis in Thonny Log Analyzer

Type of analysis	**Description**
Duplicate files	Identifies whether submitted log files are identical. This is achieved using the CRC-32 (Cyclic Redundancy Check) algorithm, which computes file checksums. If two files have matching checksums, they are considered duplicates.
Student work analysis	Examines general characteristics that may suggest log files are incomplete. It identifies cases where the number of runs, working time, or log file size falls below a specified threshold.
Pasted to typed text %	Identifies source code files within logs where the proportion of pasted text surpasses a predefined threshold.
Identical pasted texts	Examines pasted text within logs and identifies exact matches.
Source code pasted	Searches logs for pasted text that matches source code found in other students' logs.
Source code comparison	Compares source codes within logs to identify similar ones. Similarity is determined using Dice's coefficient, which must exceed a specified threshold. For this comparison, the `compactTwoStrings` function from the `string-similarity` library is used, returning a value between 0 and 1, where 0 indicates completely different strings, and 1 signifies identical ones. Since comparing all source codes would be too time-consuming, the codes are first sorted by length, and only the nearest y source codes on either side of the selected code are analyzed. The value of y is determined using the formula: $$y = \begin{cases} 5, & \text{if } \frac{10000}{x} < 5 \\ \frac{10000}{x}, & \text{if } \frac{10000}{x} \geq 5 \end{cases}$$ where x represents the length of the source code array.

it because most of the program had been pasted. A manual review confirmed that the entire solution was copied. Another case emerged in the mandatory university course IP1. The new tool verified that three students had shared their code. This occurred during an exam retake for those who had not passed it initially. A total of ten students submitted their solutions for the second attempt. The VPL similarity analysis showed high similarity between two student pairs, yet no concrete proof

of plagiarism was found in the comparative analysis. While the first pair showed strong similarity, the new tool did not detect it. However, the new tool flagged the second pair, along with a third student whose final code differed enough to remain undetected in the VPL similarity results. Despite the differences in their final programs, analysis of the logs revealed that all three had used the same exam solution. In every case, the similarity reached 100%. The shared code was also found by comparing the pasted segments with the source code in students' logs. Furthermore, one of the three students was flagged by the pasted text percentage analysis, while the other two had used the same code without copying and pasting it directly.

5. DISCUSSION

This chapter presents the main discussion of the results based on the created research model and posed research questions. A more in-depth discussion of the findings can be found in Publications I–VI. Subchapter 5.1 focuses on solver types based on the programming process analysis, variations in midterm exam scores, and the persistence of solver types during the introductory programming course. Subchapter 5.2 deals with log-based feedback and its influence on students' performance. Last subchapter 5.3 describes the options of using programming-process data to help detect plagiarism.

5.1. Solver Types and Their Differences in Performance

In this study, different solver types were identified based on the analysis of the programming process. Two different ways of categorizing solvers were applied. Figure 17 summarizes the solver types found, their distinguishing behavior features in programming, and aspects that influence performance. The figure is based on the research model described in subchapter 3.1. In the figure, the solver types are distinguished by color based on performance; two-colored groups mark differences in the group's performance among beginners and non-beginners. This subsection first discusses essential aspects of the solver types, then the differences in performance.

Solvers can be categorized into four clusters based on their behavior patterns in programming, which are determined by the following features: the frequency of code executions, the number of error messages encountered, the proportion of code written before the first run, and the rate of syntax errors. The first cluster is distinguished by a high number of code executions and error messages; the second cluster stands out due to a high rate of syntax errors; the third cluster shows a balanced profile across all measured features; and the fourth cluster is notable for initiating code runs later in the programming process. The first cluster which executed programs a lot of times and had a high number of error messages, likely used frequent trial-and-error attempts, which is a behavior that is often noted by other researchers (Jemmali et al., 2020; López-Pernas & Saqr 2021; Michaeli & Romeike, 2019). It can be said that previous research has focused heavily on investigating and highlighting the role of error messages while dealing with novices and their habits. In this light, the interesting finding is the fourth cluster, which ran their program for the first time only after writing a substantial portion of the code. This behavior has been under-emphasized, although it has been noticed that some students prefer to write a significant amount before refining it (Hosseini et al., 2014) or sometimes write code without checking its correctness (Ardimento et al., 2022). However, this behavior has not been thoroughly researched and has not been included in student profiling by cluster analysis. Interestingly, this also aligns with another classification in this thesis, which was found using a different

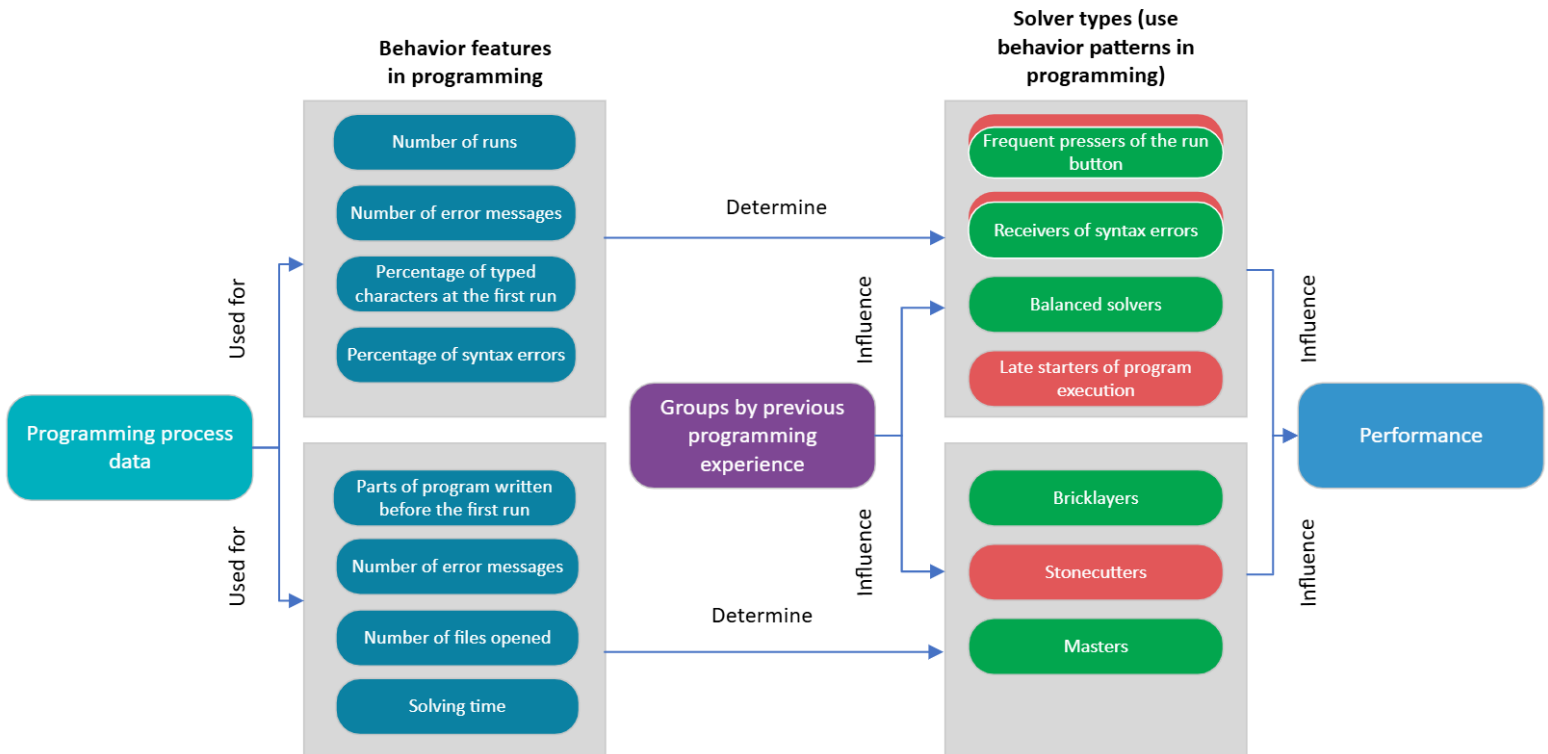

Figure 17. Solver types and their performance

approach and a smaller sample size. This classification identifies groups labeled as “Bricklayers”, “Stonecutters”, and “Masters”. The primary difference among these groups lay in their approach to solving the task, specifically, whether they built their solution incrementally, tested portions of their code (“Bricklayers”), or waited to run the program until most of the solution was already written (“Stonecutters” and “Masters”). In addition, solving time was considered, as well as the number of error messages, frequent error messages, subsequent repeated error messages, and the use of previous solutions. Every “Stonecutter” took longer to solve the exam task and encountered more error messages than any “Bricklayer”. It is essential to emphasize that two classifications using different methodologies identified a solver type that starts executing later. The group “Late starters of program execution” in the first classification aligns with “Stonecutters” in the second classification.

Starting the program execution later could also be connected to increased use of copying and pasting. For instance, “Stonecutters” who began running their programs later opened between 5 and 21 previous solutions, while “Bricklayers” only needed to look at 0 to 5 earlier solutions. Although another study has found that students often use copying and pasting, especially during the first weeks of a programming course (Vihavainen et al., 2014a), it has not been previously associated with the late start of program executions. Research has found that beginners who rely heavily on copying and pasting often require more code examples, whereas others benefit more from detailed step-by-step instructions (Blikstein, 2011), but this study highlighted the importance of teaching debugging and improving coding habits. It is essential to highlight the value of studying different solver types, as students with varying programming styles need different support (Blikstein, 2011; López-Pernas & Saqr 2021).

The four clusters described above are similar among beginners, non-beginners, and the whole group. The similar overall patterns can be attributed to the fact that all students were enrolled on an introductory programming course and generally lacked extensive experience. The study also aimed to investigate whether there were differences in two midterm exam scores between different solver types. Unlike in the second midterm exam, the groups had no statistically significant differences in the first midterm exam scores when analyzed for beginners and non-beginners separately. When analyzing beginners, it can be said that “Frequent pressers of the run button” and “Balanced solvers” achieved better results than others, although the differences were not statistically significant. One possible explanation is that students’ programming styles are still developing at the start of the course and therefore do not reveal precise trends. Another reason could be that the tasks in the first midterm exam are relatively simple, allowing students to complete them successfully even with less effective programming approaches. However, based on the whole group’s first midterm exam scores, it can be concluded that students with a high rate of syntax errors or those who execute programs late tend to perform worse than those who are balanced across all aspects.

Unlike in the first midterm exam, there were notable differences in the second midterm exam scores when analyzing separately for beginners and non-beginners. This may also suggest that skill gaps widen as the course progresses, highlighting the need for early support for struggling students. This idea is aligned with Zhang et al. (2023b), who found that students with strong debugging skills outperformed others on the first exam, with the performance gap growing even larger by the second exam. The results of the second midterm exam for both beginners and non-beginners show that the late start of the first run describes one group of students who tend to have lower scores. Additionally, among beginners, a high rate of syntax errors and a delayed start of program execution, combined with a small number of executions, are both patterns linked to weaker performance. This is an important insight, especially since weaker students have typically been associated with a high number of trial-and-error attempts (Bey & Champagnat, 2022; Blikstein, 2011; Heinonen et al., 2014; Hosseini et al., 2014; Michaeli & Romeike, 2019; Pereira et al., 2020; Tabanao et al., 2011). However, in addition to previously described weaker students, the newly identified group of struggling students does not have many error messages because they do not run their programs enough or do not do it at all. Some students attempt to write large code sections without testing or using a debugger when needed. Although weaker students are mainly associated with many error messages, it has been noted that some do not execute programs (Carter & Hundhausen, 2017), but this behavior has not been given separate attention. These findings underline the importance of teaching program debugging early in the course and adopting strategies that encourage frequent testing. Advising students to run their programs after completing small sections can be beneficial. Teaching assistants can also model good practices by regularly demonstrating how to use program executions during coding. Emphasizing this approach in the first weeks of the course can help students detect errors more effectively and deepen their understanding of how programs function.

Interestingly, among beginners, a higher rate of syntax errors is linked to poorer performance, while for non-beginners it is associated with one of the better-performing groups. Although this new finding requires further study, it suggests that performance is influenced not just by the rate of syntax errors but also by students' ability to correct them quickly and effectively (Pereira et al., 2020; Watson et al., 2013; Zhang et al., 2023b). Therefore, it is possible that non-beginners with a higher proportion of syntax errors can fix them quickly. This may be related to the observation that some students have a good attitude for taking time before designing solutions but still have many syntax errors (Bey & Champagnat, 2022). The fact that two different groups of non-beginners outperformed the other two groups highlights that there is no single approach that defines successful students. This suggests that teaching should accommodate the idea that students can effectively use different working styles.

The study also sought to detect how consistently solver types remain in the same groups during the introductory programming course among beginners and

non-beginners. The results indicate that group membership is not persistent — more students shifted to a different group than remained in their original one, regardless of whether they were beginners or non-beginners. The causes of these group changes require additional research. Further study is also needed to understand which students remain in the same group, which ones switch, and how these moves relate to their performance in the course. Previous research dealing with changes over time in behavior patterns in programming is contradictory, suggesting that changes may indicate better (Blikstein et al., 2014) or worse performance (Bey et al., 2019).

5.2. Log-Based Feedback and Its Influence on Performance

An experiment was used to evaluate the influence of log-based feedback. Logs were used to obtain data about the programming process, both for providing feedback to students and getting more information about exam performance, in addition to exam scores. Figure 18 summarizes the influence of log-based feedback on students' performance. The figure is based on the research model presented in subchapter 3.1.

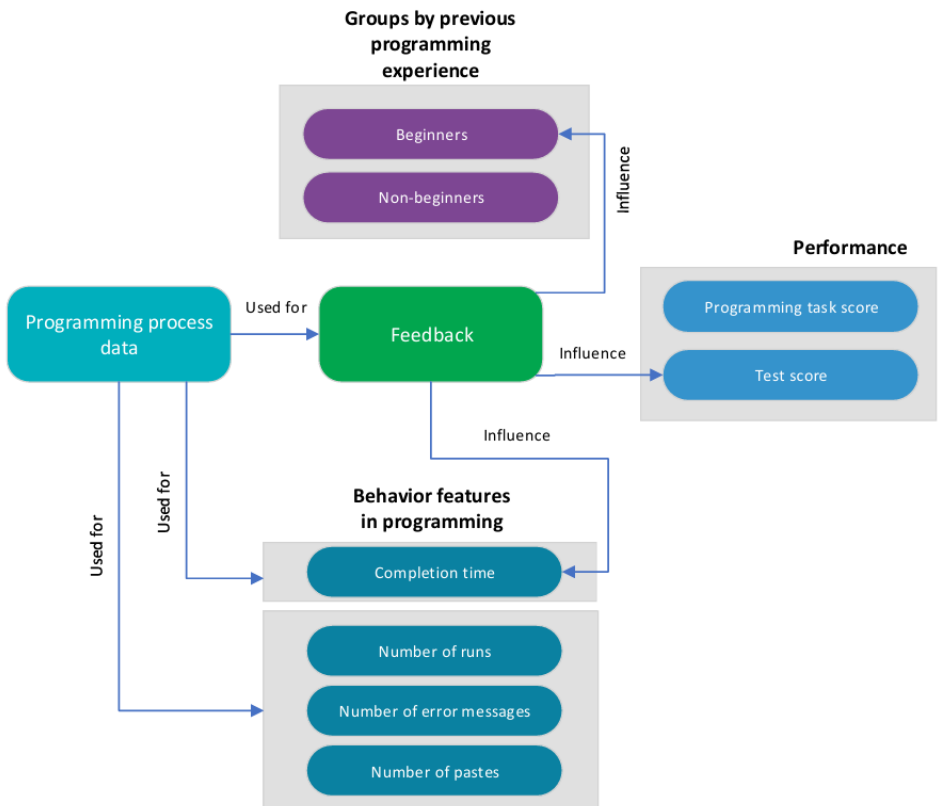

Figure 18. Influence of log-based feedback on performance

The analysis revealed that the experimental group scored significantly higher on the exam test than the control group. Although their performance on the programming task was also better, the difference was not statistically significant. Although other researchers have reported that feedback on the programming process can improve outcomes (e.g., Yan et al., 2019), this experiment added the knowledge that log-based feedback has an influence on exam test performance. This part of the exam primarily assessed students' ability to understand what a program does. The observed improvement in this skill may be linked to the nature of the feedback, which was focused on refining the programming process. For example, when logs revealed that students were making random changes in an attempt to fix their code, they were encouraged to use Thonny's debugger to grasp better how their program functioned. Thonny's debugger, which enables learners to follow the program's execution step-by-step, helped facilitate this understanding. Previous studies have also emphasized the importance of teaching debugging (Ahmadzadeh et al., 2005; Chmiel & Loui, 2004; Deeb & Hickey, 2021). However, our study clarified that regularly encouraging students to use the debugger likely accelerated the development of their code-reading skills.

In this study, beginners and non-beginners were also separately analyzed. In the experimental group, exam test scores were higher among both beginners and non-beginners, but the difference was statistically significant only for beginners. This suggests that log-based feedback was particularly beneficial for students without prior programming experience. This further emphasizes that the effectiveness of feedback can vary depending on the learner's skill level (Worsley & Blikstein, 2013) and that previous programming experience can influence students' performance in programming courses (Leinonen et al., 2016; Veerasamy et al., 2018; Zhang et al., 2013).

Several specific aspects were examined to gain a deeper understanding of how students approached the exam programming task: the time taken to complete the task, the number of program executions, error messages encountered, and instances of code pasting. The results indicated that the experimental group completed the task more quickly than the control group, with the difference being statistically significant. When breaking down the analysis by prior programming experience, it was found that both beginners and non-beginners in the experimental group completed the task faster. However, the difference reached statistical significance only for beginners. This suggests that log-based feedback notably affected beginners' task completion time. These findings are consistent with earlier research (Deeb & Hickey, 2021; Yan et al., 2019), but these results emphasize especially the importance of focusing on those novices who have never tried programming. The reason for the bigger influence on beginners may be due to the fact that they are still developing their programming habits and often use inefficient strategies, such as frequent copy-pasting, trial-and-error attempts, or avoiding executions. As a result, they benefit more from feedback that emphasizes the programming process.

The feedback based on the logs focused on improving coding practices — for example, encouraging debugging tools and promoting a methodical approach where learners write and debug code in stages to ensure each part functions correctly. This process helps them learn how to debug their programs, understand how they operate, and identify mistakes more efficiently, ultimately speeding up their ability to write functional code. Previous studies have highlighted that novices often struggle to locate errors early in their programming education (Denny et al., 2012; Marceau et al., 2011). More experienced students, on the other hand, may already have established routines that are harder to adjust. This points to the need for tailored support: beginners and non-beginners benefit from different types of guidance. Helping students focus on the programming process can reduce inefficient trial-and-error behavior and the tendency to avoid executing code, affecting task completion time.

It is crucial to recommend that educators emphasize teaching effective programming techniques to beginners and provide personalized support to those whose current approach is ineffective, for instance, those who rely heavily on trial and error or rarely run their programs. Such support can help students develop a clearer understanding of program behavior. This research shows that one effective strategy is using tools that allow instructors to observe how students write and debug code. It is essential to emphasize the role of the instructor who uses the program-process information for teaching, not just tools. The instructor's role in giving log-based feedback has been emphasized in some works (Yu et al., 2023; Zhang et al., 2023a) but not specifically in the context of developing efficient programming behaviors, encouraging debugging tools, etc. To further study log-based feedback and its efficiency, the experiment could be repeated with a larger participant group to determine the differences better.

5.3. Plagiarism Detection Based on Programming Process

To find potential plagiarism detection options for use in history-based tools, the logs were used to analyze style features in the programming process during the two programming courses using Thonny's functionality of replaying the programming process. Figure 19 summarizes the usage of suitable types of behavior features in programming for plagiarism detection.

During the analysis, style features were divided into two groups: (1) general style features; (2) programming style features. General style features describe common patterns that emerge during programming, such as the typical order in which elements are typed. Programming style features refer to the individual decisions students make when coding, especially in situations where there are several correct ways to write syntax. The findings suggest that general style features are more consistent over time. These stable patterns can help detect whether a student has completed a task independently and can be used to check plagiarism automatically. However, since students may share similar characteristics, it is essential

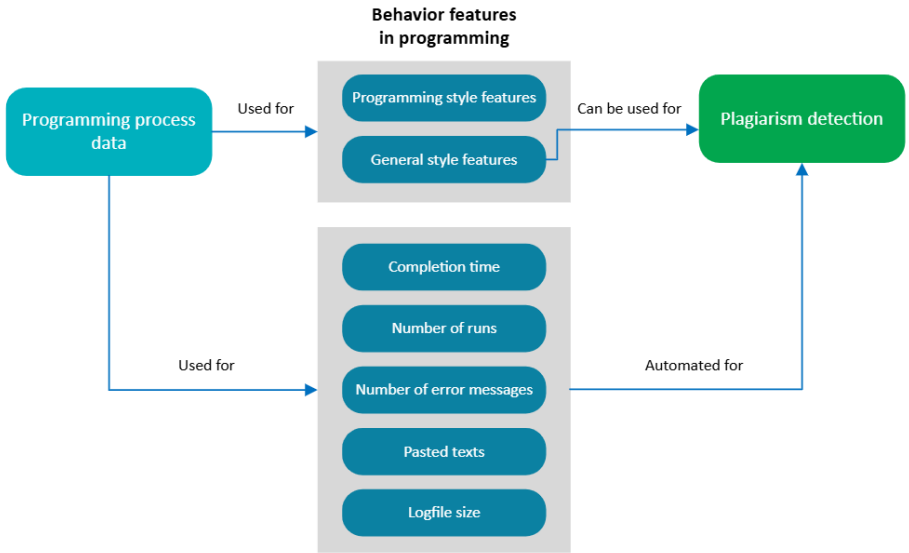

Figure 19. Types of behavior features in programming for detecting plagiarism instances

to analyze various style features and combine them with additional indicators, such as the average time it takes to type character pairs (digraphs). Some environments already use keyboard press and release events for automatic plagiarism detection (Byun et al., 2020). General style features can also be integrated with techniques that analyze code similarity. In contrast, programming style features are more variable and evolve throughout the course. Students in introductory programming classes typically do not demonstrate stable use of these features, as their coding habits are heavily influenced by examples found in learning materials. Solutions of final assignments are often shaped by the structure and style of exercises students use as examples while coding. Therefore, programming style features alone are unreliable for automatically verifying whether a student completed the task independently. On the other hand, as students now often use AI-generated code, programming style features are becoming important in another way. Namely, counting style anomalies is one possible option to help detect the use of AI-generated code (Denzler et al., 2024).

Although some IDEs have the functionality of replaying the programming process (Annamaa, 2015) or can use a plugin that records keystroke-level data (Hart et al., 2023), it is still time-consuming if the system requires the instructor to deal with every log separately. Therefore, it is essential to develop tools that enable a visualization of code-changing patterns and results of different types of analysis (Shrestha et al., 2022). It is also important to analyze many logs at once.

The plagiarism detection tool Thonny Log Analyzer, which received input from this study, uses a comparison of logs, which means incorporating programming process information into plagiarism detection. It employs six types of analy-

sis to detect potential plagiarism: it detects duplicate log files; checks overall characteristics that might indicate incomplete logs; flags source code files where the amount of pasted content exceeds a set threshold; analyzes pasted segments in the logs to find exact duplicates; searches for pasted content that matches code from other students' logs; and compares source code across logs to identify similarities between them. This tool processes logs simultaneously and visually displays the results of the different analysis types.

The following behavior features in programming, which were used in Thonny Log Analyzer, help detect incomplete logs and potential plagiarism instances: number of runs, number of error messages, task completion time, logfile size, and pasted texts. Lack of executions and error messages, or only a small number of them, may suggest that the program was not properly tested. Sometimes it can be an indicator of plagiarism. At the same time, a short task completion time and small logfile size could point to possible plagiarism or the student's submission of incorrect or incomplete files. It has also been noted earlier that the task-solving speed and the occurrence of pastes help detect plagiarism (Ljubovic & Pajic, 2020). However, copying and pasting does not automatically indicate plagiarism, as students may copy content from course materials or their own earlier work. It has been emphasized that some students copy and paste a lot while programming (Blikstein, 2011; Vihavainen et al., 2014a). Nevertheless, careful review can help identify actual cases of plagiarism if pastes are extracted automatically. Various comparisons that examine pasted content within programming-process data or even duplicate content in different students' logs, which is not pasted, help detect plagiarism that might be difficult to uncover using other approaches. These comparisons also enable instructors to identify potential cases from large amounts of data efficiently.

It is important to continue investigating style features using quantitative methods that help distinguish between students' coding patterns. This can support the development of tools aimed at detecting potential plagiarism, especially when students complete final assignments at home and instructors cannot directly observe who is doing the work.

6. CONCLUSION AND IMPLICATIONS

This chapter presents the conclusions along with theoretical and practical implications, outlines the limitations of the study, and offers recommendations for future research. The main approach was quantitative, aiming to explore how programming-process data can be utilized to support students in introductory programming courses and detect instances of plagiarism. The data was used to form and analyze solver types, investigate their variations, and persistence; to give log-based feedback and study its influence on students' performance; and to find plagiarism checking options based on programming process information.

This research showed that solvers could be categorized into groups based on their behavior patterns in programming, with similar patterns observed among both beginners and non-beginners. A key finding was the identification of a new behavior feature in programming that is commonly linked to lower performance – the late start of the first execution of the program. Specific patterns associated with higher performance were identified, and it was found that some groups achieved comparable results despite exhibiting different behaviors, thereby highlighting the diversity among learners. It was also discovered that solver types were not persistent throughout the first programming course. In addition, the study showed that providing feedback based on programming log data significantly enhanced exam test performance. Programming-process data is also valuable for developing more effective plagiarism detection options. Namely, this thesis found that general style features are a possible option for history-based plagiarism detection tools, because of their persistence. In addition, the approach relying on students' programming-process data was the foundation for creating a tool to detect plagiarism.

6.1. Theoretical Implications

Theoretical implications of this research are presented below.

The relationship between the late start of program execution and performance. This thesis identified a new behavior feature in programming associated with lower performance. Specifically, it was revealed that the late start of program execution is related to poorer outcomes in both beginner and non-beginner groups. It is essential to note that the revealed feature is not characterized by the frequency or number of errors, which is the usual understanding of features that distinguish low performers. This finding provides an essential aspect for future research exploring student performance in programming from multiple perspectives.

The importance of log-based feedback. Providing feedback based on the programming process is a relatively recent approach. So far, only a few studies have explored incorporating insights from the programming process into practical classroom activities. This study demonstrated that feedback derived from programming logs resulted in notable improvements in exam performance and

reduced the time for beginners to complete programming tasks. Therefore, this method appears particularly effective for novice learners. This is an essential piece of knowledge to consider in future studies that are investigating feedback options for beginners.

Suitable behavior features in programming for detecting plagiarism. Using programming-process information for plagiarism detection is a relatively new approach. There has been little analysis of the types of features that can be used for this purpose. The study showed that, unlike programming style features, general style features are stable and well-suited for plagiarism detection. This is essential knowledge to consider when analyzing various combinations of potential features for plagiarism detection tools.

6.2. Practical Implications

Practical implications are provided below that could be considered when teaching programming in introductory courses.

Executing programs frequently and using debuggers. The study found a connection between the late start of the program execution and students' lower performance. A key recommendation is to teach programming in a way that encourages frequent program execution as a natural and regular part of the coding process. Teachers could provide tailored exercises for students less inclined to run their code to practice programming using executions. The findings also highlight the importance of emphasizing debugging skills during the initial weeks of a course. Early focus on debugging can help reduce the number of students who either avoid running their programs or wait too long to do so. Using built-in debuggers within programming environments can significantly help students identify errors. These tools assist with debugging and enhance understanding of the program flow and the sequence of command execution. Introducing and promoting these tools is especially valuable in large classes where they can help students become more independent and methodical in their programming practice. It is also beneficial for teaching assistants to stress, particularly in the early weeks, that even small programs are composed of multiple parts. Assignments that involve building programs step-by-step can reinforce this concept. Additionally, teachers can demonstrate good practices by showing how to write programs and regularly execute them. In some cases, they can explain that students who delay executing their programs until most of the code is already written often achieve lower grades.

Considering solver types in practical sessions. The study found that certain groups achieved similar performance levels despite exhibiting different behavior patterns in programming, highlighting the diversity among learners. Insights into various behavior patterns in programming can provide teachers with valuable guidance for developing and applying diverse support strategies. Some beginners benefit more from additional examples, while others require more thorough and

detailed instructions. In particular, paying attention to the differences between solver types can sometimes be beneficial during practical sessions, and suitable practices can be identified through discussion.

Applying log-based feedback to strengthen code reading skills. The study demonstrated that log-based feedback produced the greatest improvement in the test performance in an exam that tested code reading skills. Therefore, this is a good method to use to improve code reading abilities. In the age of AI, more attention should definitely be paid to developing code-reading skills.

Using log-based feedback in teaching programming to beginners. The experiment showed that log-based feedback significantly reduced beginners' time to complete programming tasks and overall influenced more beginners. As such, this feedback approach should be used particularly for students with no previous programming experience at the beginning of an introductory programming course to get better results. To reduce teachers' workload, it is reasonable to give log-based feedback to students with ineffective behavior patterns in programming, for example, those who use the trial-and-error approach or do not run their programs.

Combining general style features with additional data to detect plagiarism. The study concluded that general style features could be an applicable choice for history-based plagiarism detection tools due to their persistence over time. On the contrary, programming style features tend to vary and evolve throughout the course. However, because some students use some general style features similarly, it is essential to use a diverse set of these features and supplement them with additional data, such as the average time to type digraphs. General style features can also be integrated with techniques that analyze program similarity.

Incorporating a history-based method for plagiarism detection. Currently, most plagiarism detection tools depend on comparing source code, which can be ineffective against obfuscation strategies used by students. As a result, a tool was developed that uses a history-based method for plagiarism detection, enabling it to identify certain types of plagiarism that might otherwise be hidden through code manipulation.

6.3. Limitations

It should be noted that this thesis has limitations that should be considered when interpreting or generalizing the results.

Constraints of log data. Collecting data from logs provides detailed insights into activities within the programming environment, but not beyond it. To gain a more comprehensive understanding in the future, it could be beneficial to include additional data from other learning environments used during the course, such as the learning management system.

Slight variation in students' academic background. The data used to find solver types were gathered from a university course primarily attended by computer science majors. While there were some groups of students from other

disciplines, the results might vary if the dataset included a wider range of academic backgrounds. Since the experiment of giving log-based feedback involved a course attended by students of various disciplines, the results might differ in a course composed primarily of computer science students.

Small granularity in an analysis. The persistence of solver types was analyzed only at a general level. A small sample did not allow for a detailed analysis of persistence and movement of solver types, nor did it include relations to performance.

Limited sample size and short courses. The primary constraints in examining style features for plagiarism detection are the limited sample size and the short duration of the courses. The findings may differ if the course is aimed at a different audience or if it is not an introductory-level course.

6.4. Suggestions for Future Work

The following are proposed suggestions for future research.

Incorporating supplementary data from other course platforms. Log data provides detailed insights into activities within the programming environment, but does not capture what happens outside of it. To gain a more comprehensive understanding in the future, it may be beneficial to include additional data from other platforms used during the course, such as the learning management system. In addition, it is a valuable future direction to incorporate more background information, which can uncover new relations that help make choices in teaching, for example, forming groups in large courses or developing personalized approaches.

Investigating the persistence of solver types in more depth. For future research, it is necessary to use a larger sample size and a more in-depth investigation into the persistence of solver types and their relationship to course outcomes. Understanding which tendencies are more or less effective throughout the course is crucial. Identifying movement patterns between groups linked to higher or lower exam scores would be valuable, as it could lead to practical strategies for enhancing teaching effectiveness. It is also uncertain whether solver types remain impersistent only during the initial programming course or over a longer timeframe. Studying changes over an extended period can provide new insights into understanding solver types and may give possibilities to help learners in a more personalized way.

Verifying results and developing a tool for automated log-based feedback. It is crucial to verify whether an experiment providing log-based feedback yields consistent results with a larger sample size and a different programming language context. A dedicated tool should be developed to streamline and automate the extraction of relevant information from logs to enable the systematic use of the method across various instructors. This tool could be designed for teachers and students, ensuring both groups receive meaningful and actionable feedback. Im-

plementing AI in this kind of educational tool certainly provides additional opportunities for analysis, feedback, and more personalization.

Enhancing the plagiarism detection tool. Looking ahead, there are many possibilities for improving the plagiarism detection tool. Enhancements could include adding new types of analysis to provide a more thorough comparison of students' development processes and to improve the accuracy of plagiarism detection. For example, integrating a timeline analysis that compares the timelines of students' work could offer valuable insights. The source code comparison analysis could also be refined using language-specific similarity algorithms. Additionally, future improvements should focus on detecting AI-generated code.

BIBLIOGRAPHY

Ahadi, A., R. Lister, and D. Teague (2014). "Falling Behind Early and Staying Behind When Learning to Program". In: *PPIG*. Vol. 14. URL: https://www.ppig.org/files/2014-PPIG-25th-Ahadi.pdf.

Albluwi, I. and J. Salter (2020). "Using static analysis tools for analyzing student behavior in an introductory programming course". In: *Jordanian Journal of Computers and Information Technology (JJCIT)* 6.3, pp. 215–233.

Alqadi, B. S. (2024). "Enhancing Novice Programmers' Debugging Skills through Systematic Education: A Comparative Study". In: *IEEE Access* 12, pp. 181192–181204. URL: https://ieeexplore.ieee.org/document/10772104.

Anjali, V., T. R. Swapna, and B. Jayaraman (2015). "Plagiarism detection for java programs without source codes". In: *Procedia Computer Science* 46, pp. 749–758. DOI: 10.1016/j.procs.2015.02.143.

Annamaa, A. (2015). "Thonny, a Python IDE for Learning Programming". In: *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 343–343. DOI: 10.1145/2729094.2754849.

Ardimento, P., M. L. Bernardi, M. Cimitile, D. Redavid, and S. Ferilli (2022). "Understanding coding behavior: an incremental process mining approach". In: *Electronics* 11.3, p. 389. DOI: 10.3390/electronics11030389.

Bennedsen, J. and M. E. Caspersen (2019). "Failure rates in introductory programming: 12 years later". In: *ACM Inroads* 10.2, pp. 30–36. DOI: 10.1145/3324888.

Bey, A., M. Pérez-Sanagustín, and J. Broisin (2019). "Unsupervised automatic detection of learners' programming behavior". In: *European Conference on Technology Enhanced Learning*. Springer, Cham, pp. 69–82. DOI: 10.1007/978-3-030-29736-7_6.

Bey, A. and R. Champagnat (2022). "Analyzing Student Programming Paths using Clustering and Process Mining". In: *Proceedings of the 14th International Conference on Computer Supported Education*, pp. 76–84. DOI: 10.5220/0011077300003182.

Blikstein, P. (2011). "Using learning analytics to assess students' behavior in open-ended programming tasks". In: *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, pp. 110–116. DOI: 10.1145/2090116.2090132.

Blikstein, P., M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller (2014). "Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming". In: *Journal of the Learning Sciences* 23.4, pp. 561–599. DOI: 10.1080/10508406.2014.954750.

Byun, J., J. Park, and A. Oh (2020). "Detecting contract cheaters in online programming classes with keystroke dynamics". In: *Proceedings of the Seventh ACM Conference on Learning@ Scale*, pp. 273–276. DOI: 10.1145/3386527.3406726.

Carter, A. S., C. D. Hundhausen, and O. Adesope (2015). "The normalized programming state model: Predicting student performance in computing courses based on programming behavior". In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pp. 141–150. DOI: 10.1145/2787622.2787710.

Carter, A. S. and C. D. Hundhausen (2017). "Using programming process data to detect differences in students' patterns of programming". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 105–110. DOI: 10.1145/3017680.3017785.

Castro, M. D. B. and G. M. Tumibay (2021). "A literature review: efficacy of online learning courses for higher education institution using meta-analysis". In: *Education and Information Technologies* 26, pp. 1367–1385. DOI: 10.1007/s10639-019-10027-z.

Chen, H. M., B. A. Nguyen, C. R. Dow, N. L. Hsueh, and A. C. Liu (2022). "Exploring Time-Related Micro-Behavioral Patterns in a Python Programming Online Course". In: *Journal of Information Science & Engineering* 38.6.

Dehalwar, K. and S. N. Sharma (2024). "Exploring the Distinctions between Quantitative and Qualitative Research Methods". In: *Think India Journal* 27.1, pp. 7–15.

Deeb, F. A., A. DiLillo, and T. J. Hickey (2018). "Using Fine Grained Programming Error Data to Enhance CS1 Pedagogy". In: *CSEDU (1)*, pp. 28–37. DOI: 10.5220/0006666400280037.

Deeb, F. A. and T. Hickey (2021). "Reflective debugging in Spinoza V3.0". In: *Proceedings of the 23rd Australasian Computing Education Conference*, pp. 125–130. DOI: 10.1145/3441636.3442313.

Denny, P., A. Luxton-Reilly, and E. Tempero (2012). "All syntax errors are not equal". In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. ACM, pp. 75–80. DOI: 10.1145/2325296.2325318.

Denzler, B., F. Vahid, A. Pang, and M. Salloum (2024). "Style anomalies can suggest cheating in CS1 programs". In: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, pp. 381–387. DOI: 10.1145/3626253.3635519.

Đurić, Z. and D. Gašević (2013). "A source code similarity system for plagiarism detection". In: *The Computer Journal* 56.1, pp. 70–86. DOI: 10.1093/comjnl/bxs018.

Ebrahimi, A. (2012). "How does early feedback in an online programming course change problem solving?" In: *Journal of Educational Technology Systems* 40.4, pp. 371–379. DOI: 10.2190/et.40.4.c.

Edwards, J., J. Leinonen, and A. Hellas (2020). "A study of keystroke data in two contexts: Written language and programming language influence predictability of learning outcomes". In: *Proceedings of the 51st ACM Technical Sympo-*

sium on Computer Science Education, pp. 413–419. DOI: 10.1145/3328778.3366863.

Estey, A. and Y. Coady (2016). "Can interaction patterns with supplemental study tools predict outcomes in CS1?" In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 236–241. DOI: 10.1145/2899415.2899428.

Fu, Q., Y. Zheng, M. Zhang, L. Zheng, J. Zhou, and B. Xie (2023). "Effects of different feedback strategies on academic achievements, learning motivations, and self-efficacy for novice programmers". In: *Educational Technology Research and Development*, pp. 1–20. DOI: 10.1007/s11423-023-10223-2.

Fujiwara, K., K. Fushida, H. Tamada, H. Igaki, and N. Yoshida (2012). "Why novice programmers fall into a pitfall?: Coding pattern analysis in programming exercise". In: *2012 Fourth International Workshop on Empirical Software Engineering in Practice*. IEEE, pp. 46–51. DOI: 10.1109/iwesep.2012.13.

Guo, P. J. (2015). "Codeopticon: Real-time, one-to-many human tutoring for computer programming". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pp. 599–608. DOI: 10.1145/2807442.2807469.

Hart, K., C. Mano, and J. Edwards (2023). "Plagiarism Deterrence in CS1 Through Keystroke Data". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pp. 493–499. DOI: 10.1145/3545945.3569805.

Heinonen, K., K. Hirvikoski, M. Luukkainen, and A. Vihavainen (2014). "Using codebrowser to seek differences between novice programmers". In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pp. 229–234. DOI: 10.1145/2538862.2538981.

Hellas, A., J. Leinonen, and P. Ihantola (2017). "Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating". In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 238–243. DOI: 10.1145/3059009.3059065.

Hellas, A., P. Ihantola, A. Petersen, V. V. Ajanovski, M. Gutica, T. Hynninen, and S. N. Liao (2018). "Predicting academic performance: a systematic literature review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pp. 175–199. DOI: 10.1145/3293881.3295783.

Herrera, G., M. Nuñez-del-Prado, J. G. L. Lazo, and H. Alatrista (2019). "Through an agnostic programming languages methodology for plagiarism detection in engineering coding courses". In: *2019 IEEE World Conference on Engineering Education (EDUNINE)*. IEEE, pp. 1–6. DOI: 10.1109/edunine.2019.8875802.

Holanda, M., L. R. De Miranda, F. Macedo, J. L. Yamin, C. Dorea, C. Chamon, and D. Da Silva (2023). "Automatic Formative and Motivational Feedback

Personalized for Introductory Programming Course". In: *2023 IEEE Frontiers in Education Conference (FIE)*. IEEE, pp. 1–7. DOI: 10.1109/fie58773.2023.10343336.

Holden, E. and E. Weeden (2003). "The impact of prior experience in an information technology programming course sequence". In: *Proceedings of the 4th Conference on Information Technology Curriculum*, pp. 41–46. DOI: 10.1145/947121.947131.

Hosseini, R., A. Vihavainen, and P. Brusilovsky (2014). "Exploring problem solving paths in a Java programming course". In: *Proceedings of Psychology of Programming Interest Group Annual Conference, Brighton, UK, 25–27 June 2014*, pp. 65–76.

Hrkút, P., M. Ďuračík, Š. Toth, and M. Meško (2023). "Current Trends in the Search for Similarities in Source Codes with an Application in the Field of Plagiarism and Clone Detection". In: *2023 33rd Conference of Open Innovations Association (FRUCT)*. IEEE, pp. 77–84. DOI: 10.23919/fruct58615.2023.10143064.

Hundhausen, C. D., D. M. Olivares, and A. S. Carter (2017). "IDE-based learning analytics for computing education: a process model, critical review, and research agenda". In: *ACM Transactions on Computing Education (TOCE)* 17.3, pp. 1–26. DOI: 10.1145/3105759.

Jadud, M. C. (2006). "Methods and tools for exploring novice compilation behaviour". In: *Proceedings of the Second International Workshop on Computing Education Research*, pp. 73–84. DOI: 10.1145/1151588.1151600.

Jain, A. K. (2010). "Data clustering: 50 years beyond K-means". In: *Pattern Recognition Letters* 31.8, pp. 651–666. DOI: 10.1016/j.patrec.2009.09.011.

Jemmali, C., E. Kleinman, S. Bunian, M. V. Almeda, E. Rowe, and M. S. El-Nasr (2020). "MAADS: Mixed-methods approach for the analysis of debugging sequences of beginner programmers". In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Portland, OR, USA, pp. 86–92. DOI: 10.1145/3328778.3366824.

Jeuring, J., H. Keuning, S. Marwan, D. Bouvier, C. Izu, N. Kiesler, and S. Sarsa (2022). "Towards Giving Timely Formative Feedback and Hints to Novice Programmers". In: *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education*, pp. 95–115. DOI: 10.1145/3571785.3574124.

Keuning, H., J. Jeuring, and B. Heeren (2018). "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises". In: *ACM Transactions on Computing Education (TOCE)* 19.1, pp. 1–43. DOI: 10.1145/3231711.

Koss, I. and R. Ford (2013). "Authorship is continuous: Managing code plagiarism". In: *IEEE Security & Privacy* 11.2, pp. 72–74. DOI: 10.1109/msp.2013.26.

Leinonen, J., K. Longi, A. Klami, and A. Vihavainen (2016). "Automatic Inference of Programming Performance and Experience from Typing Patterns". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 132–137. DOI: 10.1145/2839509.2844612.

Leinonen, J., L. Leppänen, P. Ihantola, and A. Hellas (2017). "Comparison of time metrics in programming". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pp. 200–208. DOI: 10.1145/3105726.3106181.

Leinonen, J., F. E. V. Castro, and A. Hellas (2022). "Time-on-task metrics for predicting performance". In: *ACM Inroads* 13.2, pp. 42–49. DOI: 10.1145/3478431.3499359.

Li, Z., Y. Zhang, Y. Liu, Y. Wu, and S. Wu (2023). "Applying Coding Behavior Features to Student Plagiarism Detection on Programming Assignments". In: *Journal of Circuits, Systems and Computers*, p. 2350286. DOI: 10.1142/S0218126623502869.

Ljubovic, V. and E. Pajic (2020). "Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories". In: *IEEE Access* 8, pp. 96505–96514. DOI: 10.1109/access.2020.2996146.

Lokkila, E., A. Christopoulos, and M. J. Laakso (2023). "A data-driven approach to compare the syntactic difficulty of programming languages". In: *Journal of Information Systems Education* 34.1, pp. 84–93.

Longi, K., J. Leinonen, H. Nygren, J. Salmi, A. Klami, and A. Vihavainen (2015). "Identification of programmers from typing patterns". In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pp. 60–67. DOI: 10.1145/2828959.2828960.

López-Pernas, S. and M. Saqr (2021). "Bringing synchrony and clarity to complex multi-channel data: A learning analytics study in programming education". In: *IEEE Access* 9, pp. 166531–166541. DOI: 10.1109/access.2021.3134844.

Luxton-Reilly, A. (2016). "Learning to program is easy". In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 284–289. DOI: 10.1145/2899415.2899432.

Luxton-Reilly, A., I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, and C. Szabo (2018). "Introductory programming: a systematic literature review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITICSE*, pp. 55–106. DOI: 10.1145/3293881.3295779.

Marceau, G., K. Fisler, and S. Krishnamurthi (2011). "Measuring the effectiveness of error messages designed for novice programmers". In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM, pp. 499–504. DOI: 10.1145/1953163.1953308.

Marwan, S., G. Gao, S. Fisk, T. W. Price, and T. Barnes (2020). "Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science". In: *Proceedings of the 2020 ACM Conference*

on International Computing Education Research. ACM, pp. 194–203. DOI: 10.1145/3372782.3406264.

Matsuzawa, Y., K. Okada, and S. Sakai (2013). "Programming process visualizer: A proposal of the tool for students to observe their programming process". In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ACM, pp. 46–51. DOI: 10.1145/2462476.2462493.

Medeiros, R. P., G. L. Ramalho, and T. P. Falcão (2018). "A systematic literature review on teaching and learning introductory programming in higher education". In: *IEEE Transactions on Education* 62.2, pp. 77–90. DOI: 10.1109/te.2018.2864133.

Michaeli, T. and R. Romeike (2019). "Current status and perspectives of debugging in the k12 classroom: A qualitative study". In: *2019 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, pp. 1030–1038. DOI: 10.1109/educon.2019.8725282.

Modiba, P., V. Pieterse, and B. Haskins (2016). "Evaluating plagiarism detection software for introductory programming assignments". In: *Proceedings of the Computer Science Education Research Conference 2016*, pp. 37–46. DOI: 10.1145/2998551.2998558.

Munson, J. P. and J. P. Zitovsky (2018). "Models for early identification of struggling novice programmers". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pp. 699–704. DOI: 10.1145/3159450.3159476.

Narciss, S. (2008). "Feedback Strategies for Interactive Learning Tasks". In: *Handbook of Research on Educational Communications and Technology*. Routledge, pp. 125–143.

Narciss, S. (2013). "Designing and Evaluating Tutoring Feedback Strategies for Digital Learning". In: *Digital Education Review* 23, pp. 7–26.

Narciss, S. and K. Huth (2006). "Fostering Achievement and Motivation with Bug-Related Tutoring Feedback in a Computer-Based Training for Written Subtraction". In: *Learning and Instruction* 16.4, pp. 310–322. DOI: 10.1016/j.learninstruc.2006.07.003.

Naveed, M. S. (2024). "Pedagogical suitability: A software metrics-based analysis of Java and Python". In: *International Journal of Innovation in Science and Technology* 6.4, pp. 1956–1967.

Novak, M., M. Joy, and D. Kermek (2019). "Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review". In: *ACM Transactions on Computing Education* 19.3, pp. 1–37. DOI: 10.1145/3313290.

O'Malley, C. and A. Aggarwal (2020). "Evaluating the Use and Effectiveness of Ungraded Practice Problems in an Introductory Programming Course". In: *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pp. 177–184. DOI: 10.1145/3373165.3373185.

Pereira, F. D., S. C. Fonseca, E. H. Oliveira, A. I. Cristea, H. Bellhäuser, L. Rodrigues, and L. S. Carvalho (2021). "Explaining Individual and Collective Programming Students' Behavior by Interpreting a Black-Box Predictive Model". In: *IEEE Access* 9, pp. 117097–117119. DOI: 10.1109/access.2021.3105956.

Pereira, F. D., E. H. Oliveira, D. B. Oliveira, A. I. Cristea, L. S. Carvalho, S. C. Fonseca, and S. Isotani (2020). "Using learning analytics in the Amazonas: understanding students' behaviour in introductory programming". In: *British Journal of Educational Technology* 51.4, pp. 955–972. DOI: 10.1111/bjet.12953.

Pereira, F. D., E. H. Oliveira, D. Fernandes, and A. Cristea (2019). "Early performance prediction for CS1 course students using a combination of machine learning and an evolutionary algorithm". In: *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*. Vol. 2161. IEEE, pp. 183–184. DOI: 10.1109/icalt.2019.00066.

Piech, C., M. Sahami, D. Koller, S. Cooper, and P. Blikstein (2012). "Modeling how students learn to program". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, pp. 153–160. DOI: 10.1145/2157136.2157182. URL: https://doi.org/10.1145/2157136.2157182.

Porter, L. and D. Zingaro (2014). "Importance of early performance in CS1: two conflicting assessment stories". In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pp. 295–300. DOI: 10.1145/2538862.2538912.

Rocha, H. J. B., P. C. D. A. R. Tedesco, and E. D. B. Costa (2023). "On the use of feedback in learning computer programming by novices: a systematic literature mapping". In: *Informatics in Education* 22.2, p. 209. DOI: 10.15388/infedu.2023.09.

Rodriguez-Rivera, G., J. Turkstra, J. Buckmaster, K. LeClainche, S. Montgomery, W. Reed, and J. Lee (2022). "Tracking large class projects in real-time using fine-grained source control". In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*, pp. 565–570. DOI: 10.1145/3478431.3499389.

Rodríguez-Veliz, M., Y. Nuñez-Musa, and R. Sepúlveda-Lima (2023). "Study of Code Obfuscation Techniques for the Security of Software Components". In: *International Journal of Intelligent Systems and Applications in Engineering* 11.10s, pp. 913–922.

Rubinstein, A., N. Parzanchevski, and Y. Tamarov (2019). "In-depth feedback on programming assignments using pattern recognition and real-time hints". In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 243–244. DOI: 10.1145/3304221.3325552.

Ryman, D., P. K. Imbrie, and J. Kastner (2021). “Application of source code plagiarism detection and grouping techniques for short programs”. In: *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, pp. 1–7. DOI: 10.1109/fie49875.2021.9637268.

Ryman, D., P. K. Imbrie, and J. Kastner (2022). “Sensitivity Preservation and Precision of Plagiarism Detection Engines for Modified Short Programs”. In: *2022 ASEE Annual Conference & Exposition*. DOI: 10.18260/1-2--40868.

Santos, Á., A. Gomes, and A. Mendes (2013). “A taxonomy of exercises to support individual learning paths in initial programming learning”. In: *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE, pp. 87–93. DOI: 10.1109/fie.2013.6684794.

Sharma, K., K. Mangaroska, H. Trætteberg, S. Lee-Cultura, and M. Giannakos (2018). “Evidence for programming strategies in university coding exercises”. In: *Lifelong Technology-Enhanced Learning: 13th European Conference on Technology Enhanced Learning, EC-TEL 2018, Leeds, UK, September 3–5, 2018, Proceedings 13*. Springer International Publishing, pp. 326–339. DOI: 10.1007/978-3-319-98572-5_25.

Shi, H., Y. Zhou, V. P. Dennen, and J. Hur (2023). “From unsuccessful to successful learning: profiling behavior patterns and student clusters in Massive Open Online Courses”. In: *Education and Information Technologies*, pp. 1–32. DOI: 10.1007/s10639-023-12010-1.

Schleimer, S., D. S. Wilkerson, and A. Aiken (2003). “Winnowing: local algorithms for document fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 76–85. DOI: 10.1145/872757.872770.

Shrestha, R., J. Leinonen, A. Hellas, P. Ihantola, and J. Edwards (2022). “Code-process charts: Visualizing the process of writing code”. In: *Proceedings of the 24th Australasian Computing Education Conference*, pp. 46–55. DOI: 10.1145/3511861.3511867.

Song, D., H. Hong, and E. Y. Oh (2021). “Applying computational analysis of novice learners’ computer programming patterns to reveal self-regulated learning, computational thinking, and learning performance”. In: *Computers in Human Behavior* 120, p. 106746. DOI: 10.1016/j.chb.2021.106746.

Tabanao, E. S., M. M. T. Rodrigo, and M. C. Jadud (2011). “Predicting at-risk novice Java programmers through the analysis of online protocols”. In: *Proceedings of the Seventh International Workshop on Computing Education Research*, pp. 85–92. DOI: 10.1145/2016911.2016930.

Veerasamy, A. K., D. D’Souza, R. Lindén, and M. J. Laakso (2018). “The impact of prior programming knowledge on lecture attendance and final exam”. In: *Journal of Educational Computing Research* 56.2, pp. 226–253. DOI: 10.1177/0735633117707695.

Ventura, P. and B. Ramamurthy (2004). “Wanted: CS1 students. No experience required”. In: *ACM SIGCSE Bulletin* 36.1, pp. 240–244. DOI: 10.1145/971300.971387.

Vihavainen, A., J. Helminen, and P. Ihantola (2014a). “How novices tackle their first lines of code in an IDE: Analysis of programming session traces”. In: *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pp. 109–116. DOI: 10.1145/2674683.2674692.

Vihavainen, A., M. Luukkainen, and P. Ihantola (2014b). “Analysis of source code snapshot granularity levels”. In: *Proceedings of the 15th Annual Conference on Information Technology Education*, pp. 21–26. DOI: 10.1145/2656450.2656473.

Villamor, M. M. (2020). “A review on process-oriented approaches for analyzing novice solutions to programming problems”. In: *Research and Practice in Technology Enhanced Learning* 15.1, pp. 1–23. DOI: 10.1186/s41039-020-00130-y.

Watson, C., F. W. Li, and J. L. Godwin (2013). “Predicting performance in an introductory programming course by logging and analyzing student programming behavior”. In: *2013 IEEE 13th International Conference on Advanced Learning Technologies*. IEEE, pp. 319–323. DOI: 10.1109/icalt.2013.99.

Watson, C. and F. W. Li (2014). “Failure rates in introductory programming revisited”. In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, pp. 39–44. DOI: 10.1145/2591708.2591749.

Wiedenbeck, S., D. Labelle, and V. N. Kain (2004). “Factors affecting course outcomes in introductory programming”. In: *PPIG*, p. 11.

Worsley, M. and P. Blikstein (2013). “Programming pathways: A technique for analyzing novice programmers’ learning trajectories”. In: *International Conference on Artificial Intelligence in Education*. Springer, pp. 844–847. DOI: 10.1007/978-3-642-39112-5_127.

Yan, L., A. Hu, and C. Piech (2019). “Pensieve: Feedback on coding process for novices”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, pp. 253–259. DOI: 10.1145/3287324.3287483.

Yu, J. H., X. Z. Chang, W. Liu, and Z. Huan (2023). “An online integrated programming platform to acquire students’ behavior data for immediate feedback teaching”. In: *Computer Applications in Engineering Education* 31.3, pp. 520–536. DOI: 10.1002/cae.22596.

Zhang, X., C. Zhang, T. F. Stafford, and P. Zhang (2013). “Teaching introductory programming to IS students: The impact of teaching approaches on learning performance”. In: *Journal of Information Systems Education* 24.2, pp. 147–155.

Zhang, A. G., Y. Chen, and S. Oney (2023a). “VizProg: Identifying Misunderstandings By Visualizing Students’ Coding Progress”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pp. 1–16. DOI: 10.1145/3544548.3581516.

Zhang, Y., L. Paquette, J. D. Pinto, Q. Liu, and A. X. Fan (2023b). "Combining latent profile analysis and programming traces to understand novices' differences in debugging". In: *Education and Information Technologies* 28.4, pp. 4673–4701. DOI: 10.1007/s10639-022-11343-7.

Zhi, R., S. Marwan, Y. Dong, N. Lytle, T. W. Price, and T. Barnes (2019). "Toward data-driven example feedback for novice programming". In: *Proceedings of the 12th International Conference on Educational Data Mining*. Montreal, Canada, pp. 218–227.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisors, Marina Lepp and Eno Tõnisson, for their unwavering support and guidance. Eno's example, extraordinary personality, talent, and teaching skills gave me the impetus to start teaching programming and research in the same area. I learned a lot about teaching from him. He believed in me from the beginning and was always very supportive and trusting. Marina's strong personality, systematic and practical approach, and exceptional ability to notice important details have been a great role model for me, and I am very grateful for her dedication in supervising me. I am also thankful for our collaboration in teaching and for her good advice over several years.

I would like to thank the Didactics of Informatics research group. I learned from all of you, from your discussions and experiences, and even from the emotions of those of you, who finished before me. The writing camps of our research group included valuable sharings of experiences, for which I am very grateful, in addition to the main purpose of the camps – writing. Realizing that others were struggling with similar challenges motivated me to keep going. I particularly thank Marili, with whom we started together and discussed many things, especially in the beginning of the doctoral studies and at the final stage.

My warmest gratitude goes to my husband Kuldar, whose strong support, encouragement, patience, and great unexpected jokes inspired me especially in the final part of the writing. He helped me to create a perfect balance between work and other activities, which helped me to stay motivated. Many thanks to all my relatives and friends, and warm greetings to those who occasionally asked how my doctoral thesis was going – especially to my Dad, who started many phone calls with this question. Special greetings to my sister Elle, with whom we gathered in childhood detailed information about weather, plants, words of our own language, etc., in a book we titled as Science Notebook.

The degree study has been financially supported by the Institute of Computer Science of the University of Tartu and the IT Academy funding program. This work was also supported by the Estonian Research Council grant "Developing human-centric digital solutions" (TEM-TA120).

SISUKOKKUVÕTE

Programmeerimisprotsessi andmete kasutamine sissejuhatavatel programmeerimiskursustel: lahendajatüüpide leidmine, tagasiside andmine ja plagiaadi tuvastamine

Aja jooksul on programmeerimisest kujunenud oluline oskus, mida üha enam õpivad erineva taustaga üliõpilased paljudelt erialadelt. Kuna suur osa õppimisest toimub väljaspool traditsioonilist klassiruumi, tuleb õpetamismeetodeid kohandada ning kasutada sobivaid rakendusi, et neid kursusi tõhusalt toetada. Üliõpilaste erinev varasem programmeerimiskogemus suurendab mitmekesisust veelgi. Lisaks on suur väljalangevus sissejuhatavatel programmeerimiskursustel jätkuvalt tõsine probleem. Kursuse esimesed nädalad on eriti suure tähtsusega, kuna nendest sõltub oluliselt edasine õnnestumine. Kõiki neid tegureid arvestades on oluline välja selgitada, kuidas üliõpilaste lähenemised erinevad programmeerimise õppimisel, ning leida tõhusaid viise nende toetamiseks, eriti kursuse alguses.

Selle doktoritöö eesmärk oli välja selgitada, kuidas saab programmeerimisprotsessi andmeid kasutada üliõpilaste lahendajatüüpide leidmiseks, nende toetamiseks sissejuhatavatel programmeerimiskursustel ja plagiaadijuhtumite tuvastamiseks. Andmeid koguti mitmelt kursuselt. Nendeks olid Tartu Ülikooli kursused „Programmeerimine“, „Programmeerimise alused“, „Programmeerimise alused II“ ja „Tehnoloogia tarbijast loojaks“. Andmeid kasutati lahendajatüüpide leidmiseks ja analüüsimiseks, nende erinevuste ja püsivuse uurimiseks. Samuti oli oluline logipõhise tagasiside andmine ja väljaselgitamine, kuidas sedalaadi tagasiside mõjutab üliõpilaste tulemusi. Programmeerimisprotsessi andmeid kasutati ka plagiaadituvastuseks sobivate tunnuste leidmiseks. Täiendavaid andmeid koguti kursuste eelküsitluste kaudu, et selgitada varasema programmeerimiskogemuse olemasolu, samuti saadi infot Moodle'i õppekeskkonnast ja õppeinfosüsteemist (ÕIS). Peamine lähenemisviis oli kvantitatiivne. Klasterdamist kasutati peamise meetodina lahendajatüüpide moodustamiseks ja analüüsimiseks. Selleks, et saada teada, kui tõhus on programmeerimisprotsessi logide põhine tagasiside, viidi läbi eksperiment.

Uurimistöö näitas, et üliõpilasi saab jaotada iseloomulikesse lahendajatüüpidesse programmeerimise käitumismustrite põhjal ning mustrid on sarnased nii algajate kui ka mittealgajate seas. Peamise liigituse aluseks olid järgmised tunnused: käivitamiste arv, veateadete arv, süntaksivigade osakaal ja kirjutatud tähemärkide osakaal programmi esimesel käivitamisel. Peamised tuvastatud lahendajatüübid olid: 1) “Sagedased käivitajad”, 2) “Süntaksivigade tegijad”, 3) “Tasakaalukad lahendajad” ning 4) “Hilised käivitajad”. Oluline tulemus on hilise programmi käivitamisega alustamise seos madalamate kontrolltöö tulemustega. Täpsemalt selgus, et hilisem käivitamisega alustamine on seotud madalamate tulemustega nii algajate kui ka mittealgajate hulgas. See on oluline tulemus, kuna väga palju on madalamaid tulemusi seostatud veateadete sagedusega, aga on ole-

mas ka selline madalamate tulemustega grupp, kes ei saa palju veateateid, kuna ei käivita programme või teevad seda vähe. Kõrgemate kontrolltöö tulemustega on leitud lahendajatüüpidest "Tasakaalukad lahendajad", keda iseloomustavad kõikide kaasatud tunnuste osas keskmisest madalamad näitajad. Samas suutsid mõned lahendajatüübid saavutada sarnaseid tulemusi hoolimata erinevatest programmeerimise käitumismustritest, mis viitab mitmekesisusele. Oluline on märkida, et erinevalt esimesest kontrolltööst ilmnesid gruppide vahel suuremad erinevused teise kontrolltöö tulemustes, mis võib viidata sellele, et oskuste tasemete erinevused kasvavad kursuse jooksul. Uuring näitas ka, et kuulumine lahendajatüüpidesse ei ole kursuse kestel püsiv.

Selgus, et programmeerimisprotsessi logide põhjal antud tagasiside parandas märgatavalt kontrolltöö testi tulemusi, mis kontrollib koodi lugemise oskust. Eksperiment näitas, et logide põhjal antud tagasiside vähendas oluliselt just algajate programmeerimisülesannete lahendamiseks kuluvat aega ning parandas nende kontrolltöö testi tulemusi. Programmeerimisprotsessi andmed osutusid väärtuslikuks ka plagiaadituvastuse viiside täiustamisel. Uuring tõi esile, et üldised stiilitunnused – näiteks teatud sümbolite kirjutamise järjekord – võivad oma püsivuse tõttu olla tõhusad plagiaadi tuvastamiseks programmeerimisprotsessi andmete põhjal. Programmeerimiskeele süntaksiga seotud valikud aga ei ole püsivad, kuna sõltuvad palju õppematerjalides olevatest näidetest. Kontrolltöö lahendused on tugevalt mõjutatud ka näidislahendustest, mida üliõpilased programmeerimise ajal kasutavad. Programmeerimisprotsessi andmeid kasutav lähenemine võeti aluseks ka rakenduse väljatöötamisel, mis võrdleb üliõpilaste logidest saadud infot, et tuvastada võimalikke plagiaadijuhtumeid.

Doktoritöö tulemuste põhjal on oluline soovitus kasutada õpetamisel lähenemist, mis julgustab üliõpilasi oma programme sageli käivitama, nii et see oleks regulaarne ja loomulik osa koodi kirjutamise protsessist. Õppejõud võiksid praktikumides demonstreerida, kuidas programmeerida nii, et käivitamine on töö käigus regulaarselt kasutusel. Tulemused rõhutavad samuti silumise oskuste arendamise olulisust kursuse esimestel nädalatel. Keskendumine silumisele võib aidata vähendada nende üliõpilaste arvu, kes püüavad programme kirjutada ilma neid käivitamata. Selle uurimistöö tulemuste põhjal on algajate koodi lugemise oskuste parandamiseks tõhus kasutada programmeerimisprotsessi logidel põhinevat regulaarset tagasisidet. See on eriti oluline tehisintellekti ajastul, mil tugevate koodi lugemise oskuste arendamine on üha vajalikum.

PUBLICATIONS

CURRICULUM VITAE

Personal data

Name:	Heidi Taveter (née Meier)
Date of Birth:	21.06.1977
Citizenship:	Estonia
Contact:	heidi.taveter@ut.ee
ORCID:	0000-0002-2638-7253

Education

2020–2026	University of Tartu, Computer Science (Doctoral Studies)
2016–2018	University of Tartu, Conversion Master in IT (MSc)
2006–2011	Baltic Methodist Theological Seminary, Theology (Diploma Studies), *cum laude*
2000–2003	Tallinn Pedagogical University, Estonian Philology (Master by Research)
1995–2000	Tallinn Pedagogical University, Teacher of Estonian Language and Literature (equal to MA); Minor: Informatics
1992–1995	Saaremaa Co-educational Gymnasium, mathematics and physics biased class
1984–1992	Kaarma Primary School

Employment

2021–...	University of Tartu, Institute of Computer Science, Junior Lecturer in Informatics
2019–2020	University of Tartu, Institute of Computer Science, Assistant in Informatics
2007–2019	Estonian Public Broadcasting, Web Editor
2003–2007	Tallinn University, Department of Estonian Philology, Lecturer
2001–2006	Tallinn University, Webmaster, Assistant to the Head
2000–2000	Tallinn Nõmme Upper Secondary School, Teacher of Estonian Language and Literature

Scientific work

Main fields of interest:

- Didactics of informatics
- Computer-assisted education

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Heidi Taveter (neiupõlvenimi Meier)
Sünniaeg:	21.06.1977
Kodakondsus:	Eesti
Kontakt:	heidi.taveter@ut.ee
ORCID:	0000-0002-2638-7253

Haridus

2020–2026	Tartu Ülikool, informaatika (PhD)
2016–2018	Tartu Ülikool, infotehnoloogia mitteinformaatikutele (MSc)
2006–2011	EMK Teoloogiline Seminar, teoloogia (diplomiõpe), *cum laude*
2000–2003	Tallinna Pedagoogikaülikool, eesti filoloogia (MA, teaduskraad)
1995–2000	Tallinna Pedagoogikaülikool, eesti keele ja kirjanduse õpetaja; lisaeriala: informaatika
1992–1995	Saaremaa Ühisgümnaasium, matemaatika-füüsika süvaõppega klass
1984–1992	Kaarma Põhikool

Teenistuskäik

2021–...	Tartu Ülikool, arvutiteaduse instituut, informaatika nooremlektor
2019–2020	Tartu Ülikool, arvutiteaduse instituut, informaatika assistent
2007–2019	Eesti Rahvusringhääling, veebitoimetaja
2003–2007	Tallinna Ülikool, eesti filoloogia osakond, lektor
2001–2006	Tallinna Ülikool, veebimeister, juhiabi
2000–2000	Tallinna Nõmme Gümnaasium, eesti keele ja kirjanduse õpetaja

Teadustegevus

Peamised uurimisvaldkonnad:

- Informaatika didaktika
- Arvutite kasutamine hariduses

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.
68. **Eerik Muuli.** Automating the assessment and feedback processes in IT teaching – improving creation and maintenance from the teaching staff perspective. Tartu 2025, 196 p.
69. **Kateryna Kubrak.** Towards User-Centered Prescriptive Process Monitoring Systems. Tartu 2025, 151 p.
70. **Zhigang Yin.** Computing and Sensing in a Smart Ring. Tartu 2025, 251 p.
71. **Abdul-Rasheed Olatunji Ottun.** Practical Trustworthy Artificial Intelligence with Human Oversight. Tartu 2025, 239 p.
72. **Sander Mikelsaar.** Analysis and Optimization of Iteratively Decodable Codes. Tartu 2025, 146 p.
73. **Marharyta Domnich.** Advancing Human-Centric Counterfactual Explanations in Explainable AI. Tartu 2026, 210 p.
74. **Viacheslav Komisarenko.** Aligning Training Loss to Evaluation Metrics in Deep Learning. Tartu 2026, 165 p.