Utah State University

DigitalCommons@USU

All Graduate Theses and Dissertations | Graduate Studies

8-2022

Programming Process, Patterns and Behaviors: Insights from Keystroke Analysis of CS1 Students

Raj Shrestha
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Shrestha, Raj, "Programming Process, Patterns and Behaviors: Insights from Keystroke Analysis of CS1 Students" (2022). *All Graduate Theses and Dissertations*. 8576.
https://digitalcommons.usu.edu/etd/8576

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

UtahStateUniversity
MERRILL-CAZIER LIBRARY

PROGRAMMING PROCESS, PATTERNS AND BEHAVIORS: INSIGHTS FROM KEYSTROKE ANALYSIS OF CS1 STUDENTS

by

Raj Shrestha

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

John Edwards, Ph.D. Major Professor	Chad Mano, Ph.D. Committee Member
Steve Petruzza, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2022

Copyright © Raj Shrestha 2022

All Rights Reserved

ABSTRACT

PROGRAMMING PROCESS, PATTERNS AND BEHAVIORS: INSIGHTS FROM KEYSTROKE ANALYSIS OF CS1 STUDENTS

by

Raj Shrestha, Master of Science

Utah State University, 2022

Major Professor: John Edwards, Ph.D.
Department: Computer Science

CS1 courses are the introductory programming courses for many programmers. For this reason, it is necessary for students to understand the concepts well. It is also crucial for the instructors to address misconceptions of students and help them by understanding their thought process. If we are able to identify struggling students early in the course, then effective interventions can be taken by the instructors. Keystroke data collected from students has become popular in computing education research. This type of data is quantitative, often of high temporal resolution, and it can be collected non-intrusively while the student is in a natural setting working on their assignments. This research use Keystroke data from CS1 students, analyzes them to predict struggling students, understand student behaviors and thought processes on how they are formulating their solutions. Instructors have very little insight into how students actually work on their programming assignments, as code submissions are typically a summative assessment. Through interactive visualization tools, statistical data-analysis and unsupervised learning we find evidence of keystroke data being able to identify struggling students, understand student's thought process and behavior and detect plagiarism.

The keystroke data is collected from students following university IRB protocol. We investigate the existence of chronotypes through unsupervised learning that aligns with typical populations reported in the literature, and our results support correlations of certain chronotypes to academic achievement. We also investigate pausing behavior of students, pause-frequency of different lengths, the last action before pausing, and correlations with exam scores. We find evidence that frequency of pauses of all lengths is negatively correlated with exam score, and that pausing behavior can be an indicative factor of struggling students. Additionally, we propose an interactive software tool (called Code-Process) with a novel visualization that includes both static and dynamic views of the student's programming process. It offers instructors a view into how students actually write their code and can have broad impacts on assessment, intervention, instructional design, and plagiarism detection. We also report results of an exploratory think-aloud study in which two instructors offer thoughts as to the utility and potential of the tool.

(91 pages)

PUBLIC ABSTRACT

PROGRAMMING PROCESS, PATTERNS AND BEHAVIORS: INSIGHTS FROM KEYSTROKE ANALYSIS OF CS1 STUDENTS

Raj Shrestha

With all the experiences and knowledge, I take programming as granted. But learning to program is still difficult for a lot of introductory programming students. This is also one of the major reasons for a high attrition rate in CS1 courses. If instructors were able to identify struggling students then effective interventions can be taken to help them. This thesis is a research done on programming process data that can be collected non-intrusively from CS1 students when they are programming. The data and their findings can be leveraged in understanding students' thought process, detecting patterns and identifying behaviors that could possibly help instructors to identify struggling students, help them and design better courses.

To all CS1 students

ACKNOWLEDGMENTS

This work would not have been possible without the help of a lot of people. I have a lot of respect and gratitude to all these people and I don't think these words are sufficient to express my gratitude and respect for them. But I will take this opportunity to express my deepest gratitude to my advisor Dr. John Edwards. Without his constant support, motivation and feedback this work would not exist. I would like to thank my committee members Dr. Steve Petruzza and Dr. Chad Mano for their time and advice. I would also like to extend my gratitude to Dr. Juho Leinonen, Dr. Arto Hellas, Dr. Petri Ihantola and Albina Zavgorodniaia for their thoughts, motivation and support on all the collaborative work.

I am always indebted to my parents, Amrit Shrestha and Tulasa Shrestha, who went to great lengths to educate me and my brother and taught us about hard work, dedication and patience. I would also like to thank my brother, Raju Shrestha for constantly inspiring me and motivating me through out this journey. I am blessed to be with my love of my life, Prakriti Dumaru who is constantly pushing me to go the extra mile and is always on my side through highs and lows.

I am thankful to Aashish Ghimire who constanly supports me and helps me with his suggestions and feedback. I am wholeheartedly thankful to Megh Raj KC and Bijaya Thapa who treated us like family members and supported us. I am also thankful to my friends Ankit Shrestha, Pravin Poudel and Rizu Paudel for being there and supporting me. I would also like to extend my gratitude to Nepalese Community here in Logan. Finally, I am grateful to the staff and members in the Computer Science department for their assistance during my time as a graduate student.

Raj Shrestha

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . v

ACKNOWLEDGMENTS . . . vii

LIST OF TABLES . . . x

LIST OF FIGURES . . . xi

1 Introduction . . . 1

2 CodeProcess Charts: Visualizing the Process of Writing Code . . . 3
- 2.1 Introduction . . . 3
- 2.2 Related Work . . . 4
  - 2.2.1 Code and snapshot playback . . . 5
  - 2.2.2 Structure of the code and state space . . . 5
  - 2.2.3 Code measures over time . . . 6
  - 2.2.4 Combining location and time in pixel maps . . . 7
- 2.3 CodeProcess Chart and Software . . . 8
  - 2.3.1 *CodeProcess* chart . . . 9
  - 2.3.2 Code playback . . . 13
  - 2.3.3 Final code . . . 13
- 2.4 Think-Aloud Study . . . 13
  - 2.4.1 Context and data . . . 14
  - 2.4.2 Study details . . . 14
  - 2.4.3 Student process . . . 15
  - 2.4.4 Plagiarism detection . . . 17
  - 2.4.5 Interpretability of the *CodeProcess* chart . . . 18
  - 2.4.6 Feedback to the student . . . 19
  - 2.4.7 Other results . . . 19
- 2.5 Conclusion . . . 20

3 Pausing While Programming . . . 21
- 3.1 Introduction . . . 21
- 3.2 Related Work . . . 22
  - 3.2.1 Pausing behavior . . . 23
  - 3.2.2 Pausing behavior in CER . . . 24
  - 3.2.3 Typing and performance in programming . . . 26
- 3.3 Methodology . . . 26
  - 3.3.1 Context and data . . . 27
  - 3.3.2 Event and pause categories . . . 29

3.3.3 Statistical tests . . . 30
3.4 Results . . . 30
3.4.1 Descriptive statistics . . . 30
3.4.2 Frequency of pauses . . . 31
3.4.3 Student types . . . 32
3.4.4 Initiating pauses . . . 34
3.5 Discussion . . . 36
3.5.1 Frequency of pauses . . . 38
3.5.2 Student types . . . 39
3.5.3 How pauses are initiated . . . 39
3.5.4 Threats to validity . . . 43
3.6 Conclusions . . . 43

4 Circadian Rhythms of CS1 Students . . . 46
4.1 Introduction . . . 46
4.2 Background and Related Work . . . 47
4.2.1 Circadian Rhythms and Chronotypes . . . 47
4.2.2 Chronotypes in Software Engineering . . . 49
4.3 Methodology . . . 50
4.3.1 Context and Data . . . 50
4.3.2 Research Questions . . . 52
4.3.3 Metrics . . . 52
4.3.4 Ethics . . . 53
4.4 Results . . . 54
4.4.1 Comparison of binning strategies . . . 54
4.4.2 Chronotypes learned from clustering . . . 55
4.4.3 Chronotype and relation to course outcomes . . . 57
4.4.4 Differences in context . . . 59
4.5 Discussion . . . 59
4.5.1 Robustness of clustering . . . 60
4.5.2 Robustness to external factors . . . 60
4.5.3 General Insights . . . 61
4.5.4 Effect on course outcomes . . . 62
4.5.5 Synchrony with chronotype . . . 62
4.5.6 Limitations . . . 63
4.6 Conclusion . . . 63
4.7 Data Availability . . . 65

5 Conclusion . . . 66

REFERENCES . . . 68

LIST OF TABLES

Table Page

3.1 Summary of contexts. 28

3.2 Descriptive statistics of the study. 31

3.3 Statistics of the clusters for the two contexts, US and European. For the Cluster column, "*shorter*" means "*shorter pause*" and similar with *longer*. A t-test for the two distributions of exam scores yields ($t = 2.2, p = 0.026, d = 0.35$) in the US context and ($t = 2.1, p = 0.034, d = 0.28$) in the European context. 35

3.4 Pearson r correlations with p values between a student's tendency to initiate a given length of pause with a given event type and exam score. "All" indicates percentage across all events (both those initiating pauses and not). We do not have data on tabs or whether a run was successful or not in the Java context, so tab values are not included for the Java context and the *(Success) run* column should be interpreted as a successful run for the Python context and all runs for the Java context. 37

4.1 Summary of contexts. Daylight is measured on January 1. 52

4.2 Statistics of the clusters for the two contexts, US and European. The four *Centroid* columns indicate the 4D centroid discovered in k-means clustering. *Students* is the number of students in each cluster. *Keystrokes* is the median number of keystrokes per student. *Homework average score* is on project A5 (max of 100 points) for the US context and on all homework programming assignments (max of 215 points) for the European context. *Exam average score* is on the first midterm (max of 100 points) for the US context and is a combined score on the two exams (max of 30 points combined) for the European context. *Hours before deadline* is calculated by finding the median keystroke timestamp for each week and averaged over all weeks. Standard deviations are included with the homework/exam scores and hours before the deadline averages. 55

4.3 Results of running a Kruskal-Wallis H significance test across clusters for different outcomes. 60

LIST OF FIGURES

Figure Page

2.1 *CodeProcess* software. The *CodeProcess* chart is on the left. The playback window is on the top right and the final code window is on the bottom right. 9

2.2 Visualization of submissions 4, 5, 8, and 11. All but submission 5 were used in the pilot study. 10

2.3 a Zoomed in version of submission 5 (Figure 2.2b) at events 1307-1434 and including characters 705-770. b Details on submission 4 (Figure 2.2a). . . . 11

2.4 Zoomed in view of submission 11. See Fig. 2.2d. 11

2.5 a Example of a student typing in code from another solution. b Example of what might look like a student typing code from another solution but what, in this case, is a capable student writing their code linearly. 12

3.1 Log-scale bar chart showing the total number of events for each type. a Number of events in the US/Python context. b Number of events in the European/Java context. 31

3.2 Frequency of the different type of pause correlated with exam score. Frequency is calculated as number of pauses divided by total number of events. 33

3.3 Correlations of total pauses with each other per student. As expected, the number of micro pauses a student takes has a strong positive correlation with the number of short pauses. While there are still strong and medium correlations between shorter and longer pauses, the correlations become weaker. 33

3.4 Similar to Fig. 3.3, this figure shows correlation coefficients for the different pause lengths. 34

3.5 Clustering. In the Python context, t-test statistics (t, p) and effect sizes (d) between the two distributions are: Micro $(t = 1.5, p = 0.12, d = 0.23)$, Short $(t = -6.9, p = 3.6e{-}11, d = -1.01)$, Mid $(t = -6.4, p = 8.4e{-}10, d = -0.93)$, and Long $(t = -3.2, p = 0.0014, d = -0.47)$. In the Java context, t-test statistics (t, p) and effect sizes (d) between the two distributions are: Micro $(t = 1.7, p = 0.08, d = -0.22)$, Short $(t = -10.8, p = 2.4e{-}22, d = -1.35)$, Mid $(t = -9.3, p = 1.1e{-}17, d = -1.15)$, and Long $(t = 2.1, p = 2.6e{-}9, d = -0.77)$. 35

3.6 Grouped bar chart showing normalized/relative frequencies of keystrokes by pause length. "All" are all events, whether they precede a pause or not. . . 37

4.1 Comparison of keystroke distributions for each proposed chronotype across feature vector definitions for the US dataset. Vector binning strategies are: First = [3-9, 9-15, 15-21, 21-3]; Second = [5-11, 11-17, 17-23, 23-5]; Third = [7-13, 13-19, 19-1, 1-7]. 54

4.2 Distribution of keystrokes for the four clusters for each context labeled with proposed chronotypes. 56

4.3 Outcomes by cluster: project score, exam score, hours remaining before due date, and number of keystrokes. Whiskers extend 1.5·IQR past the low and high quartiles. 58

4.4 Keystroke distribution of the evening chronotype in the US split into weekday and weekend keystrokes. 61

CHAPTER 1

Introduction

As a graduate student I often take programming for granted. But, every year on average one third of students fail their introductory programming courses [1, 2]. This is one of the major reasons for the high attrition rate of computer science students from the major. Effective teaching interventions from instructors can be of great help for students in understanding the concepts and increasing the pass rates in CS1 classes. However, as many as one quarter of the students still fail the courses after teaching interventions [3]. One of the reasons for this is ineffective interventions that are given to all students, whether they are struggling or not. If we are able to identify struggling students early in the course then effective interventions can be taken by the instructors on the targeted students.

Computing Education Research (CER) is a field of research that is concerned with learning and teaching computing. The researchers in this field seek ways to improve and teach computing effectively. Researchers use different kinds of data and techniques to learn about students' thinking process and identifying struggling students. Analysis of programming process data has become popular in this field in the last decade. Keystroke data is a type of programming process data that is quantitative, often of high temporal resolution, and it can be collected non-intrusively while the student is in a natural setting. This type of data has different granular levels and includes all the activities of students, such as submission, computation, edit and keystroke events. This data can be also used in predicting course outcomes [4–6], describing student behaviors [7–9], generalizing findings between two contexts to ensure that the results are generalizable [10], determine code authorship [11] and detect plagiarism [12].

This thesis is based on three original peer-reviewed publications reporting a research done on Keystroke data collected from CS1 students to predict struggling students, understand student behaviors and gain insight into their thought process when they are formu-

lating their solutions. Code submissions used by instructors to assess students are just a summative assessment and give no further insights. Keystroke data can be leveraged to visualize students' programming process to gain further insights on the students thought process. Chapter 2 discusses the use of Keystroke data to generate visualizations using a software tool - CodeProcess Charts and also includes the results obtained from a think-aloud study from two instructors on the usage and usability of the tool. Instructors find the visualizations to be useful in understanding their students' thought process and different approaches used by them to solve a programming assignment.

Similarly, Chapter 3 gives some insights into pausing behaviors of students when they are programming. The pausing patterns of students have correlations with academic performance of students and the pausing behavior can help identify struggling students. The final chapter is a research done on Circadian Rhythms of students. It investigates the existence of chronotypes using unsupervised learning. The chronotypes we find align with those of typical populations reported in the literature and our results support correlations of certain chronotypes to academic achievement. We also find that procrastinating students are at the risk and perform lower compared to other populations. The articles included in this thesis are outlined below:

- Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. CodeProcess Charts: Visualizing the Process of Writing Code. Twenty-Fourth Australasian Computing Education Conference (ACE). 2022.

- Raj Shrestha, Juho Leinonen, Albina Zavgorodniaia, Arto Hellas, and John Edwards. Pausing While Programming: Insights From Keystroke Analysis. In ACM International Conference on Software Engineering (ICSE), Software Engineering Education and Training (SEET) track. 2022.

- Albina Zavgorodniaia, Raj Shrestha, Juho Leinonen, Arto Hellas, and John Edwards. Morning or Evening? An Examination of Circadian Rhythms of CS1 Students. ACM International Conference on Software Engineering (ICSE), Joint Track on Software Engineering Education and Training (JSEET). Madrid, Spain. 2021.

CHAPTER 2

CodeProcess Charts: Visualizing the Process of Writing Code

2.1 Introduction

One primary aim of introductory programming courses is to teach students how to develop computer programs. However, if a student cannot attend classes, or if the instructor does not write code in the classes, or if other means to see how programs are constructed are not offered, students do not have the opportunity to observe the *process* of writing code [13]. Similarly, instructors (and automated assessment systems) are often blind to the students' process of writing code, as they often use submissions of final code to assess students' abilities in writing code [14, 15]. The problem is that the final code gives no hint as to the process that a student took to write it. Two students, one of whom sailed through development of the code, and another who may have struggled, may submit code that looks very similar. This is especially true in introductory computer programming courses, where the programming assignments can be relatively simple.

One research stream with the potential to alleviate this issue is the collection and use of intermediate snapshot data from students' computers as they write programs [16], one example of which is keystroke data [17]. It has been suggested that keystroke-level data may provide significantly more information on, e.g., what sorts of programs students try out and what sorts of syntax errors students encounter when compared to submission data or snapshots taken, e.g., when running or testing the program [18]. Most previous work on such data has focused on analyzing compilations and predicting course outcomes [4–6, 9, 19–22], understanding typing behaviors [11, 23–27], studying how novices construct programs [28], improving programming skill through code playback [29], etc. However, only little emphasis has been invested into building and using (interactive) visualizations of how students write programs.

While prior work on visualizations of code changes has mostly focused on bigger programming projects with relatively coarse change granularity [30,31], such work is less common in studying fine-grained changes, especially within the domain of seeking to understand how novices write code. Thus, in this study, we use similar ideas to those used in visualizing large software projects [32]. Moreover, we present an interactive visualization tool called *CodeProcess*[1] that allows the user to peer into how a computer program is developed. While the tool includes standard playback and file differencing functionality seen in some other programming process visualization tools (c.f. [33,34]), the primary value of the tool comes from a static chart that provides a summary of how a file was written at a glance. This chart is the centerpiece of the interactive visualization and shows the user which parts of the file were written when, as well as features like re-writing the same code, trouble spots, pastes, and refactoring. We evaluated *CodeProcess* using a qualitative, exploratory think-aloud study with external instructors, where we studied *how instructors use the CodeProcess chart and what sorts of insights they come up with from viewing the visualizations.*

This article is organized as follows. We first discuss related work, outlining previously proposed tools for analyzing the programming process. In Section 2.3, we present the *CodeProcess* tool. Section 2.4 presents the think-aloud study, and Section 2.5 gives conclusions.

2.2 Related Work

Software visualizations are used in both educational and professional context. Based on Diehl, visualizations can focus either on the structure, behaviour, or evolution of software [35]. In this study we talk about evolution of students' code. Evolutional visualizations are used for many purposes, including (professional) project management and understanding developers [31]. As objectives in education and professional software development are at least partially overlapping, we will provide examples from both.

The basis for visualizating the programming process is the possibility to collect data from students who are programming, be it programming assignment submissions data or

[1]The *CodeProcess* code is publicly available at two repositories: `github.com/EdwardsLabUSU/CodeProcess-API` is pre-processing Python code and `github.com/EdwardsLabUSU/CodeProcess-UI` is JavaScript UI code.

more fine-grained data such as keystroke data [16]. The last decades have witnessed a noticeable increase in collection and use of snapshot data from introductory programming classrooms [16, 36–40].

2.2.1 Code and snapshot playback

Programming process visualization and analysis tools at times come with code and snapshot playback functionality, which allows a view to how the software was developed. As an example, the Student Coding and Observation Recording Engine (SCORE) [41] provides a view that shows a diff-style navigation of code changes in a timestamped order over the files in an edited project. The Programming Process Visualizer (PPV) [33] also provides a source code view that allows replaying how the code under analysis was written.

While the previous two examples are desktop applications, browser-based analysis tools also exist. For example, CodeBrowser [34] provides source code snapshot playback and navigation functionality with the possibility of using a dual-view that highlights differences in each subsequent snapshot. CodeBrowser also provides functionality for tagging the displayed data for future analysis, and uses an API for retrieving the visualized data which in principle allows changing the server from where the shown data is retrieved from. Similar recording playback functionality is also provided in CSQuiz [42] which is an online programming environment that supports recording and replaying programming sessions.

In general, code and snapshot playback tools allow detailed analysis of the recorded programming processes. For example, Toll [43] observed that only approximately 15% of novice programmers time is spent writing code, while 40% of the time is spent reading and navigating code, and, when comparing the behavior of high-performing and poorly performing students, Heinonen et al. [34] observed that poorly performing students rarely had a systematic approach to solving the programming problems. Such analyses can be time-consuming for the researcher, however.

2.2.2 Structure of the code and state space

At a higher abstraction level, visualizations can use Abstract Syntax Trees (ASTs),

e.g., by highlighting how (nested) AST blocks travel during the evolution of a software [44], or in general by highlighting how the structure of the code changes over time. As a concrete example of the latter, Helminen et al. [45] and Piech et al. [46] have both demonstrated how state transition graphs illustrating the transitions between intermediate stages of solutions can provide an overview to the different solution strategies in a single programming task. The examples are in the context of visual block based programming languages where the the number of different block structures (i.e., states) is quite limited.

In the context of traditional programming languages, Piech et al. [47] have also used code snapshots at each compile and measured distance between snapshots using three metrics: bag of words, API calls, and AST change, though in the end, API calls were heavily influential and AST changes were only somewhat influential. The end product was an HMM-derived flow chart of how different groups of students developed their code, which was then used as an effective predictor of exam score.

2.2.3 Code measures over time

Some of the tools plot various aggregate statistics over time. For example, SCORE [41], PPV [33], Retina [48], and ClockIt [49] each provide an overview of the programming process using either aggregate statistics (or pixel-based visualizations discussed in the next section). These aggregate statistics include, for example, details on the number and type of compilation errors, the time that the student has spent on the project, the size of the project over time, and information on testing and running the projects. Some of the tools such as Retina also provide the possibility for students to gain an insight of the processes of other students, as well as hints on the errors to look out for and estimates on how long the project will take to complete. The same approach can be used in a professional context to foster awareness in software teams [50]. Many of these metrics used in education (e.g., code complexity [51]) are adopted from the generic software quality research.

There are tools that also focus specifically on building an overview of a project, as well as tools that work on already-generated aggregate statistics. For example, SnapViz [52] takes in tab-delimited data to build visualizations from students' programming process;

similarly, ArAl [53] takes in a file that contains the number of source code snapshots for each student when working on a particular assignment as well as a file with course outcomes, and then creates a set of aggregate variables from the data that could be of interest to teachers and researchers. These tools, on the other hand, often have the problem that there is either no way to move from the aggregate statistics to the source code, or moving from the aggregate statistics to the source code can be cumbersome.

Plots of aggregate statistics and and other characteristics of a project over time are often used together with playback or diff view tools. Programming Process Visualizer (PPV) [33], mentioned earlier, is a good example of that. Interesting changes in time-plots can be clicked to see the corresponding section in the playback view.

2.2.4 Combining location and time in pixel maps

Plotting code quality measures over time may help in identifying when something interesting has happened in the code. To this end, *CVSscan* [32] visualizations utilize pixel-map representation where one dimension is time and the other is location in the code: the horizontal axis is time in terms of commits and the vertical axis is the line number of the file. Colors of pixels illustrate the age of the last change (at the given time and location), red indicating that the line was changed at that time point. The approach makes it relatively easy to identify interesting areas (as horizontal stripes) and the map is also used to navigate in the linked code view.

As our proposed *CodeProcess* charts and CVSscan use similar ideas, we here discuss the differences and motivations behind the differences. *CVSscan* is designed for the use of the maintenance community: "the main activities a maintainer performs are related to context recovery" [32]. Accordingly, *CVSscan* processes change at a lower resolution in time and space than our *CodeProcess* charts. The time resolution for *CVSscan* is every commit to a CVS code repository, whereas *CodeProcess* charts compute changes at every keystroke. Showing changes at each commit is sufficient for the needs of software maintainers, who are looking for general context of changes in large software projects with many contributors and commits. Our purpose, however, is to visualize and understand how a single developer

writes code, for which commit-level resolution is not sufficient. Regarding spatial resolution, *CVSscan* uses each pixel to represent a line, whereas we use pixels in *CodeProcess* charts to represent a single character. Again, this relates to the goals of the visualizations: line-level resolution is sufficient for a software maintainer to intuit the context in which a change is made, but character-level changes are required to understand a student's cognitive processes.

The interpretations and linked tools between *CVSscan* and *CodeProcess* also differ. For example, *CVSscan* has no playback option, while the playback option that is linked to the *CodeProcess* chart is helpful not only in interpreting the chart (Sec. 2.4.5) but also in interpreting the actions of the student (Sec. 2.4.3). Another motivational difference is why users would zoom in to a portion of the chart. In the case of *CVSscan*, the maintainer is interested in the function of the final code surrounding a change, whereas users of *CodeProcess* charts already understand the function of the final code (they designed the assignment, after all) – they want to understand why the student made that particular change at that time. Finally, *CodeProcess* charts could potentially be used for student feedback, for which a keystroke-resolution chart and playback are fundamental features.

2.3 CodeProcess Chart and Software

CodeProcess is a software visualization tool that is designed to give the viewer an immediate assessment of the general characteristics of the process used to develop a piece of software. It features interactive controls for the user to analyze details of the process. It is a web application developed using Python, D3 and React JS.

Given keystroke logs, we first preprocess them into a file that indexes the data. Any keystroke log can be converted to our format provided it has the inserted/deleted code, information about where in the code the change occured (either a single index into linearized code or row/column pair), and a timestamp. The indexed data files are loaded in the browser-based *CodeProcess* software. The software can be viewed and experimented with at `code-analysis-e1a5d.web.app`. A short demonstration video is available at `youtu.be/ptawbgpi0HI`. There are three main windows in the software tool: the *CodeProcess* chart, the code playback window, and the final code window. See Figure 2.1.

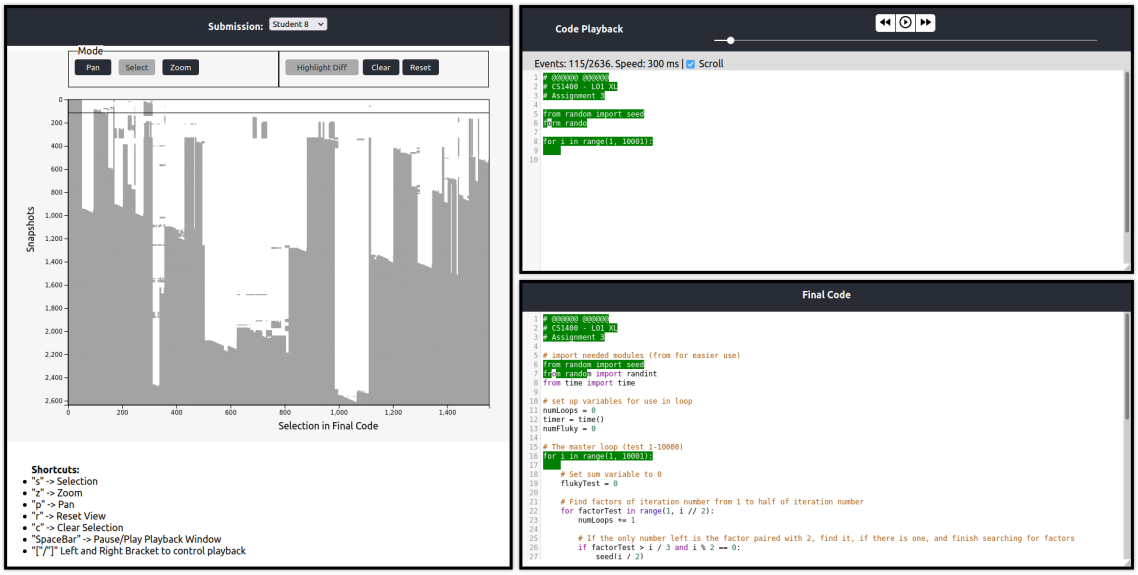
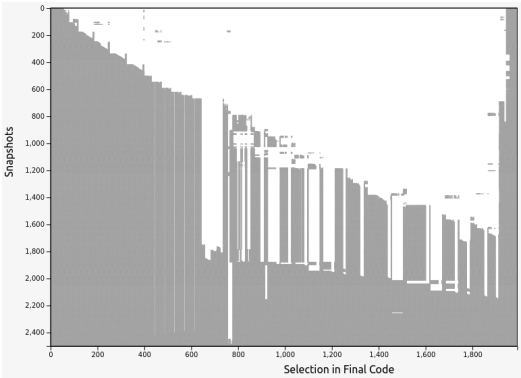


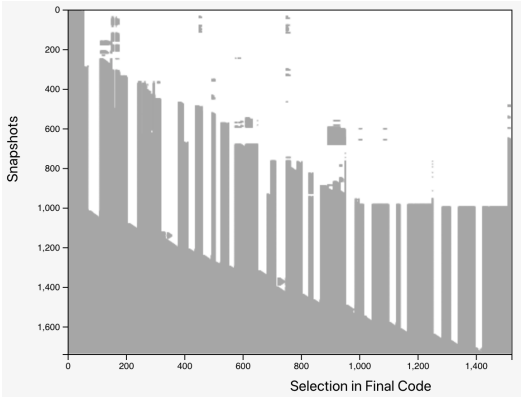
Fig. 2.1: *CodeProcess* software. The *CodeProcess* chart is on the left. The playback window is on the top right and the final code window is on the bottom right.

2.3.1 *CodeProcess* chart

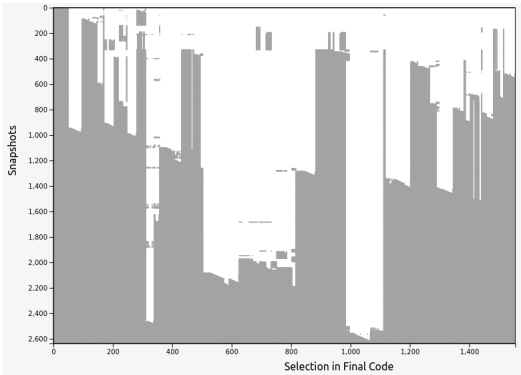
The *CodeProcess* chart is the centerpiece of the software tool. See Figure 2.2. The *CodeProcess* chart is a 2D grid with keystroke event indices on the y axis and character index of the final submission on the x axis. The x axis indexes into a linearization of the final code. A grid cell at (x, y) is colored in if the character at index x is represented in the snapshot at keystroke event y. The last row of the chart, after the final keystroke event, will have every cell filled in because, by definition, it matches the final version of the code. For example, see the solid green line in Figure 2.3a. This line is at event (i.e. keystroke) 1340 and the part of the text of the snapshot after that event is in the box outlined in solid green. Following the solid green line are a series of events, or rows in the chart, where the student types "go to next circle posi". You can see how the number of characters that match the final version of the code increase as we go down in the chart. The dashed blue line is in the middle of the student's typing. Then, at event 1372, at the tip of the triangle, the student decided to delete the text and retype it after adding "set and". Finally, at the dotted red line the student has completed the correction and the section of code matches what is in the final version. The triangle feature in the *CodeProcess* chart is an indication



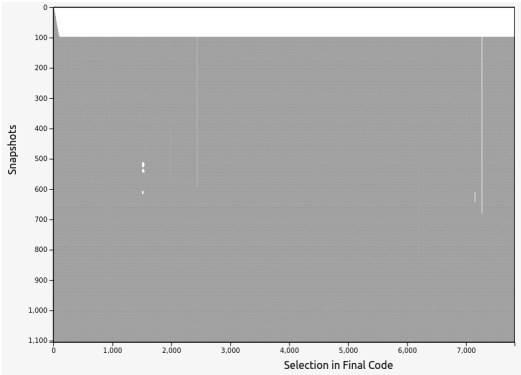
(a) Submission 4



(b) Submission 5

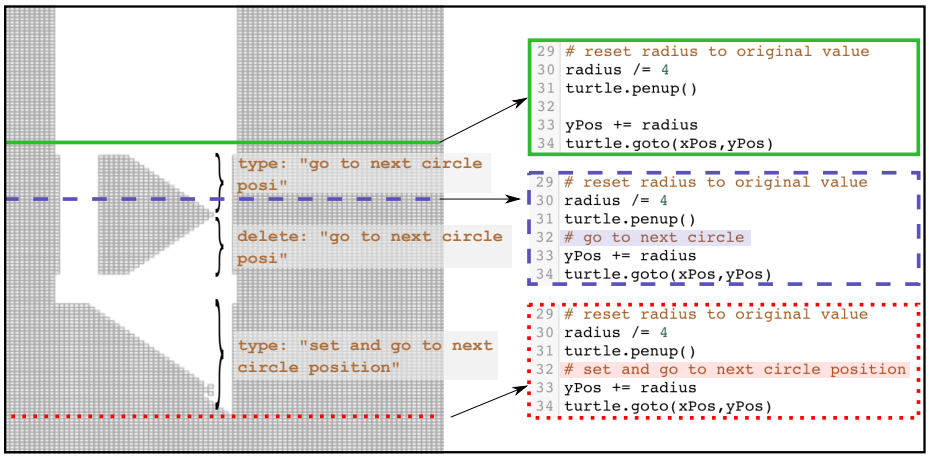


(c) Submission 8

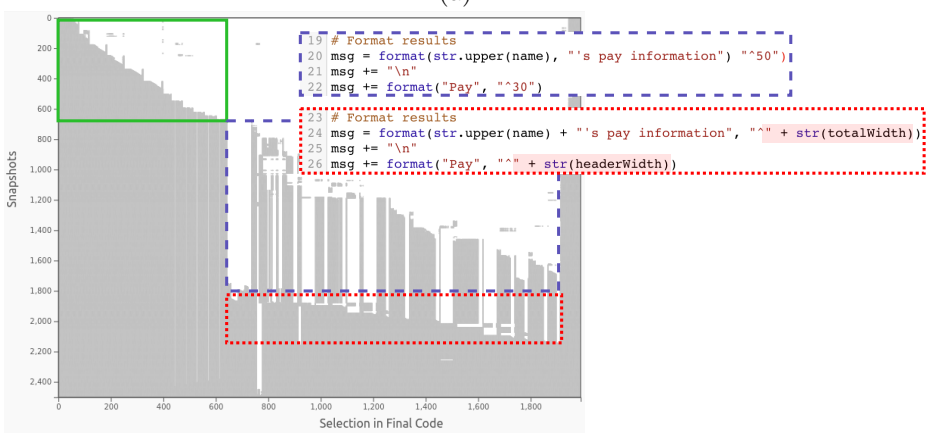


(d) Submission 11

Fig. 2.2: Visualization of submissions 4, 5, 8, and 11. All but submission 5 were used in the pilot study.



(a)



(b)

Fig. 2.3: a Zoomed in version of submission 5 (Figure 2.2b) at events 1307-1434 and including characters 705-770. b Details on submission 4 (Figure 2.2a).

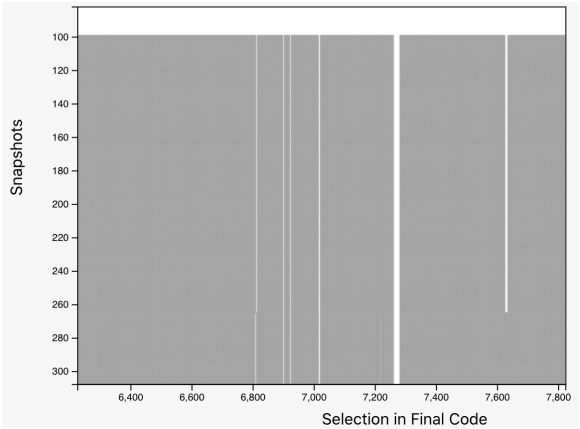
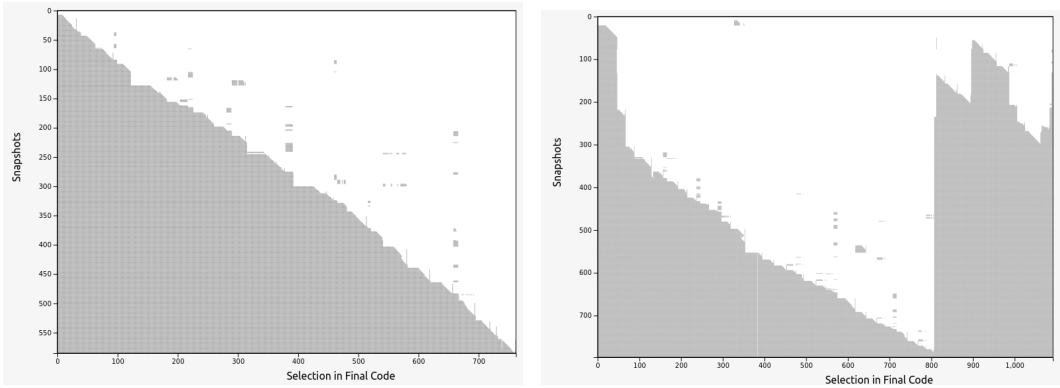


Fig. 2.4: Zoomed in view of submission 11. See Fig. 2.2d.



(a) (b)

Fig. 2.5: a Example of a student typing in code from another solution. b Example of what might look like a student typing code from another solution but what, in this case, is a capable student writing their code linearly.

that the student typed correct code then deleted it.

In Figure 2.3b we see details of submission 4. This student started out by linearly typing in straightforward variable calculations (solid green box) followed by a series of append operations to the string variable `msg` (dashed blue boxes). The student then went back and modified many of the append statements (dotted red boxes). Submission 5 (Figure 2.2b) has a similar structure – in that submission, the student wrote a series of statements and then went back and interspersed comments between them. The chart features evenly spaced "pillars" which is an indication that students wrote a number of lines of code and then went back and wrote code between them. In our data this is most often the case when students either write comments then write code under each comment (thin pillars) or when they comment the code at the end (thick pillars). It does not occur when the student comments as they go. Of course, as we saw in submission 4, pillars will appear if students do some other periodic modification to multiple lines of code.

Figure 2.4 shows a zoomed in version of submission 11. The zoomed out version of submission 11 (Figure 2.2d) looks like a solid block, indicating a large paste. Indeed, this is the case, which indicates a plagiarized submission. If we zoom in (Figure 2.4) we see thin vertical white lines. These lines indicate places where the student modified variable names and comments to mask the plagiarism.

Figure 2.5a shows an example of a student copying a solution by typing it in. In this case the student may believe that typing the solution instead of copy-pasting it will make the plagiarism less obvious, but the triangular shape of the chart makes it fairly clear what is happening. The chart in Figure 2.5b is similar but isn't completely linear. In this case the student is a capable programmer writing a merge sort. They typed the test cases and structural code first, accounting for the first block code written linearly, and then wrote the recursive function.

From the *CodeProcess* chart the user can understand when the particular section of the code was written, whether student used a top-down or bottom-up approach to formulate solution, whether they started with comments, differences between novice and expert programming patterns, and identify plagiarized solutions. The plot is also interactive and supports zoom, pan, and brush selection features.

2.3.2 Code playback

The playback window (Figure 2.1) can be used to play back the keystroke events of a student to see how the student formulated their solution. The slider allows the user to see the snapshot at a particular keystroke event (at the horizontal line shown in the chart in Figure 2.1). The snapshot of the code can then be compared to the final code using a highlight diff feature that highlights the code present in final code. The playback also has a pause/play feature and speed control buttons.

2.3.3 Final code

The final code window (Fig 1) displays the submitted solution of a student. If we highlight a particular section on the of the *CodeProcess* chart, it will highlight that section on the final code. The user can compare the final code with the snapshot and see the differences using the highlight feature.

2.4 Think-Aloud Study

In this section, we provide the results of exploratory think-aloud sessions with two instructors. We discuss the different use cases of the tool along with instructors' thoughts and their experience with the tool.

2.4.1 Context and data

We collected keystrokes in a CS1 course during Spring semester, 2021. The course was taught online (because of COVID-19) at a mid-sized public university in the United States. At the beginning of the semester students were given the opportunity to opt into the study according to the university's IRB protocol #11554, and this paper uses data only from students who opted in. The course was identical for students who chose to participate in the study and those who chose not to. 15 students volunteered for the study. Students were required to install a plugin to the *PyCharm* IDE and acknowledge that their keystrokes would be recorded. The plugin recorded keystrokes while students wrote their programming assignments. A total of 81 submissions were collected.

2.4.2 Study details

CS1 instructors from two different public universities in the United States were recruited to participate according to IRB #12110. We will use the pseudonyms Joseph and Peter. Being an exploratory, qualitative study, we did not gather quantitative data, but rather, sought to understand different approaches in how instructors might use and value the tool. The two instructors were asked to participate in a single, one-hour think-aloud session over Zoom that was recorded. At the beginning of each think-aloud session, the researcher gave a brief description of the *CodeProcess* visualization software and the participant watched a two-minute video tutorial on the usage and features of *CodeProcess*. The video tutorial did not give any guidance on interpretation of the different features of the *CodeProcess* chart. The instructors then interacted with data from three submissions: 4, 8, and 11 (see Figure 2.2). From approximately 15 candidate student submissions we chose these three submissions because they appear to represent four important behaviors in programming: linear code development (submission 4), returning to make changes to

code developed linearly (submission 4), non-linear code development (submission 8), and plagiarism (submission 11). In submission 4 the student was asked to compute net pay given gross pay, tax rate, etc. The student wrote boilerplate code linearly, then wrote a series of string appends, followed by going back and modifying many of the string appends. In submission 8 the student wrote a fluky number program. The student bounced back and forth in the code, writing different parts of the program in a seemingly random way. Submission 11 was a text-based blackjack game. The student pasted most of the assignment from elsewhere and then modified variable names, strings, and comments to mask the plagiarism. Instructors were asked to think aloud as they interacted with the tool and gained insights into students' code development process.

The two instructors initially took different approaches to the tool. Peter started right off with interpreting the *CodeProcess* chart and using playback as an auxiliary tool. Joseph initially relied primarily on the playback and paid little attention to the chart. This approach was more intuitive, but it took him longer to gain insights into student behavior because he had to watch the replay which, even when replayed at high speed, takes longer than just glancing at the chart. Eventually the researcher encouraged Joseph to spend a little time on the chart and within a few minutes he was able to link insights from the chart to the replay.

2.4.3 Student process

The visualization was useful for the instructors to understand the student's approach toward implementing a solution. The "pillars" in the chart show what part of the final code was completed first. Studying the patterns of these pillars can be useful to see if the solution was developed using a top-down (submissions 4 and 5) or bottom-up approach (submission 8), understand the student's thinking process, and identify common solution patterns. Instructors in our study were able to see if the student used linear (again, submissions 4 and 5) or non-linear thinking (submission 8) in developing a solution. Instructors also found it useful in showing when the student edited a particular section of code. After looking at submission 4 (Figure 2.2a), Peter said, "It appears that, in general, code was

generated kind of top to bottom in a linear fashion. This is a pretty straightforward assignment. It didn't require much nonlinear thinking". Similarly, Joseph thought that the student in submission 4 had a reasonably clear understanding of what needed to be done. The student was working in a linear way as if they had planned out the solution on paper before attempting the solution.

Instructors also identified that different submissions showed different approaches to getting the solution. For submission 4 (Figure 2.2a) both Peter and Joseph agreed that the student took the approach of just making the code to work and later tidying up the code in a linear way. But for submission 8 (Figure 2.2c), Peter and Joseph had different insights. Peter thought that the student was more deliberate in their approach:

> Okay, so this looks like much less linear. Which is interesting, because the nonlinear aspects of it showed up first. Maybe you can tell that the student is thinking about the solution before writing the code.

Joseph, however, thought the student didn't plan well:

> They're jumping around. Like they get going, like, "oh, yeah, I need to do this." And they go back and add that feature, like the loop counter...I feel like this program probably took them twice as long to write this way than if they would have thought it through on paper first...they probably would have got it right at the first try instead of this iterative [approach].

It is remarkable that the two instructors viewed the same student's approach with such a difference of opinion. We suggest that the difference in opinion between the two instructors is an important result that raises a number of questions: Was submission 8 effectively written or not? Are students most successful when they program linearly or not? Why would two instructors have such strong differences of opinion? We expect that instructors would have very similar opinions of the quality of final code submissions, but the differences of opinion regarding the process students took to write the code implies that we, as a community, may need to seek better understanding of what best practices for code development

process actually are. Until now we haven't had readily available tools that can effectively communicate how code evolves while a student completes an assignment. *CodeProcess* fills this gap and allows the research community to explore and potentially quantify exactly how students write their code. This is especially important because our anecdotal answers to these questions are often influenced by our teaching methods. For example, Joseph, who was concerned that the student in assignment 8 wrote code without a plan, teaches his students to first make a plan:

> The way I kind of teach my students...is, you know, sketch it out on paper a few times...and then when you translate it into code...it logically should make sense. Maybe you got a few syntax errors, but the overall structure is there for you.

Whether this is actually the best approach could be explored using *CodeProcess*. Indeed, both instructors agreed that the tool can be useful in distinguishing and characterizing their students based on different patterns of programming skills.

2.4.4 Plagiarism detection

Detecting and proving plagiarism has always been a challenging problem. Plagiarism detection methods like MOSS [54] flag submissions for a potential plagiarism based on analysis of only the final code snapshot. Due to this, prior work has suggested looking at the process instead of only the final submission to identify plagiarism [55]. *CodeProcess* allows an instructor to see how a solution was created over a time and plagiarism, when effected through pasting code or typing someone else's code, is immediately detectable. Understanding the context of plagiarism can also be helpful for instructors to identify the weak areas of students. Instructors can then help students to strengthen these weak areas and reduce plagiarism on future assignments. Submission 11 (Figure 2.2d) was plagiarized: it can be seen from the *CodeProcess* chart that the student pasted a large portion of code and then changed parts of the code to mask the plagiarism.[2] The instructors in our study

[2] We were surprised that a student would consent to a study collecting their keystrokes and then commit an egregious act of plagiarism. The student may have relied on the statement in the informed consent

(who were not the instructors of the course) were able to identify the plagiarized solution easily by looking at the visualization. This was especially evident in Peter's experience. Peter accidentally caught a glimpse of the *CodeProcess* chart for submission 11 at the very beginning of the think-aloud session – before he knew anything about the interpretation of the chart. He then watched the training video and explored submissions 4 and 8. In the middle of looking at submission 8 he remarked that the *CodeProcess* chart he had caught a glimpse of at the beginning (submission 11) must have been a case of plagiarism. Later, as he explored submission 11, he said,

> Yeah it's pretty obvious that this person copied this code from somewhere. And it looks like they're basically just changing variable names. Presumably to make it look like it's not copied. So I immediately flagged this from suspicion to just outright cheating.

Two things are worth noting in this quote: first is that Peter recognized that the student was changing variable names. The second is his use of "immediately." We envision a tool that shows the instructor a matrix of many *CodeProcess* charts at a time and hypothesize that in seconds the instructor could pick out suspected cases of cheating. A machine learning approach to detect cheating from the chart would be even better. Joseph was also able to identify plagiarism on submission 11. He also discovered cosmetic changes made by the student to mask the dishonesty. Both instructors agreed that the visualization tool can be used to generate reports of students after each assignment across the spectrum of CS classes to detect plagiarism and dishonesty. Rule-based heuristics, e.g., software that looks for large pastes, could be used to help detect plagiarism. Using *CodeProcess* charts as a confirmatory tool could allow the heuristics to have higher type I error rates.

2.4.5 Interpretability of the *CodeProcess* chart

In designing our study we were concerned that the *CodeProcess* chart might be difficult to interpret, but the instructors in our study had no trouble with interpretation, especially

document indicating that the instructor of the course would not see their keystrokes or, more likely, they may have simply forgotten that their keystrokes were being recorded.

when given a playback and highlighting tools to explore with. Peter quickly caught on and was able to identify features after only a few minutes. Joseph was initially more interested in the playback tool, but after exploring the chart for a few minutes was also able to detect and interpret features, including the "pillars" and plagiarism. By the end of their sessions, each instructor made insightful suggestions as to improving the tool: Peter suggested a matrix of charts for quick identification of suspected plagiarism, and Joseph suggested that we include run events to see if students were trying to brute-force a solution. Both instructors agreed that the tool was easily understandable and useful for CS classes.

2.4.6 Feedback to the student

Peter suggested that having students see a visualization and playback of their own code writing "would encourage them to think more deeply about [their] problem solving approach." Joseph said that he would like to use the visualization in conferences with his students, using it "as a tool I could sit down with and we could go back to their recording...where we could watch how the work actually went, that probably be more honest witness of their work than what they recall from doing it." Peter also thought that the *CodeProcess* chart would be useful to students, as "the students can also see what their main chart for the solution would look like" as compared to the chart for an expert.

2.4.7 Other results

After a think-aloud session instructors gave us some suggestions and use cases of the tool. Peter suggested that the tool can be useful in other fields too. He thinks the tool can be useful in English classes to detect plagiarism and to understand how students are formulating their essays. Peter also suggested the use of machine learning to flag students automatically and group novice and expert programmers. He also thinks a report or a summary after each submission can be useful for instructors to understand how their students are performing in their course. Both instructors agreed that automatically flagging suspected plagiarism would be useful.

2.5 Conclusion

In this work, we presented *CodeProcess* which is a novel tool for visualizing the programming process (example visualizations shown in Figure 2.2). The tool utilizes keystroke data to show in which order different parts of the source code were developed. In addition, we conducted a pilot think-aloud study to evaluate whether computing instructors can leverage the visualizations for pedagogical insights.

Our aim in developing the tool was to provide instructors with easy-to-understand visualizations that tell something about the process a student took to arrive at their solutions at a glance. We hypothesized that instructors could use the tool – for example – to augment assessment; to determine whether students are solving a programming problem in a top-down or a bottom-up manner; that the tool could be used to identify cases of plagiarism; and that the visualization could also indicate whether a student is struggling. Additionally, the tool could be used to visualize the programming process to the student themselves for reflection, or show students their peers' processes to allow students to see other solution approaches and problems other students might have had when programming. These analyses could be enhanced through first identifying specific cases – or stereotypical cases – from the data using, say, machine learning methodologies.

The results of the think-aloud study suggest that instructors are able to understand the visualizations and use the tool with little training. Both instructors interviewed in the study could identify plagiarism and recognized top-down versus bottom-up approaches taken by different students. Interestingly, the instructors interpreted one case differently. In the bottom-up process shown in Figure 2.2c, one instructor considered that the student is planning their solution, while the other hypothesized that the student did not plan well which resulted in "jumping around". This highlights that the tool could also be used to help instructors explore which solution approaches result in the best outcomes. Lastly, we suggest that the tool could also be used in the professional context for code reviews and allow professional programmers to reflect on their process.

CHAPTER 3

Pausing While Programming

3.1 Introduction

Pausing during work is a natural behaviour for a person which allows them to reflect on their task, plan what they are going to do next, revise, or take a rest. Pauses, however, can also be initiated by distraction and lead to hindering one's working process. In this paper we aim to study pauses that students take while doing programming projects in Introductory Computer Programming (CS1). In a typical CS1 course instructors and graders look at the final program a student has produced for assessment, but there is no indication from the code as to how the student wrote it. It is possible that information on the number and types of pauses students take could be mined to shed more light on processes that underlie programming in CS1.

Since pauses can be products of various activities (e.g., thinking, disengaged), we investigate whether pause length may hold insights into these activities. Our study features four types of pauses. *Micro* pauses (2-15 seconds) which may indicate the student is thinking about the code on a low level or "locally" (e.g. syntax). *Short* pauses (15 seconds to 180 seconds) may indicate that the student is involved in a higher-level process such as planning or revision. *Mid* pauses (3-10 minutes) may indicate that the student is disengaged or that they are going to an outside resource for help (e.g. YouTube, Stack Overflow, or course materials). Finally, *long* pauses (greater than 10 minutes) may indicate disengagement from the task.

In this paper, we look at relative number of pauses over the course and correlate with outcomes (exam score). That is, if a student takes more or fewer pauses relative to their total number of keystrokes, could it suggest their better or worse course performance? Many pauses that are very small may indicate that the student is planning their typing carefully

rather than writing without a clear direction and may have a positive correlation with performance. Many medium-sized pauses may indicate the same thing, but the measurement may be confounded by students who are easily distracted. Many long pauses may indicate distraction.

The research questions we investigate in this paper are:

RQ1 Is there a correlation between the relative number of pauses a student takes and their performance (exam score)?

RQ2 What groups of students exist when clustering on pausing behavior?

RQ3 What events initiate a pause and how does this correlate with the performance of the student?

We seek to answer these research questions using analysis of keystroke data collected in two CS1 courses at different universities on different continents. The closest matches to our work come from two different research streams. One of the research streams has studied syntax errors and identified pauses or breaks when correcting such errors (e.g. [38,56–59]). The other research stream has focused on the analysis of keystroke data, which has been shown to be effective at gaining insights into student behavior [17,55,60] as well as predicting student outcomes [22,24,61]. However, little work exists at the intersection of these research streams, where our work lies. The novelty of our analysis is that we are looking less at typing behavior and more at pausing behavior, which might indicate more or less of a student's cognitive processing, examining of external resources, or disengagement.

In this paper, we report on several findings: those students who pause more often generally show worse performance in the course; students who take more *shorter* pauses perform better than students who take more *longer* pauses; *mid* pauses have the strongest negative correlation with exam scores; specific events that precede pauses have a more evident correlation with performance and thus allow conjecture about the underlying processes.

3.2 Related Work

3.2.1 Pausing behavior

In the research literature, pauses are prevalently discussed in relation to language production – written or oral speech/narration [62–64], language translation [65] and editing [66]. Relying on cognitive psychology, researchers associate pauses with cognitive processing of various types [66]. For example, in writing, it is thought that pauses at higher-level text units (e.g. between sentences) are likely to be conditioned by higher-level subprocesses, such as planning and organization of the content, whereas pauses at lower-level units (e.g. between and inside words) – by lower-level subprocesses, such as morphology encoding and retrieval of items from one's memory [67].

A pause is also considered to signify cognitive effort imposed by language production mental processes [65, 68]. Butterworth1980 hypothesised that the more cognitive operations are needed for output production, the more pauses arise. damian2009advance and Revesz2017 argued that the length of a pause taken before a textual unit reflects the mental effort made with respect to production of this forthcoming unit. Reflecting on pausing in post-editing, OBrien2006 concluded that pausing patterns do, to an extent, indicate cognitive processing. However, they are ultimately subject to individual differences.

Pausing has also gained attention in the medical training domain. Lee2020 studied pauses and their relation to cognitive load. Students had to complete a medical game that simulated emergency medicine under two conditions: pause-available and pause-unavailable. In the study, pauses of two types were identified: reflection and relaxation. The first type is argued to enhance task-related cognitive processes and therefore increase mental effort (or cognitive load). The second type reflects the opposite process when the load lowers due to the resting state.

That being said, pauses during problem-solving can signify not only ongoing mental work but a suspension of it caused by various things. gould2014makes defines three types of interruptions: those that are relevant to the task and reinforce processes in the working memory, those that are relevant to the task and interrupt processes in the working memory, and those that are not relevant to the task. The author states that how these interruptions

affect the following resumption and productivity depends on "contextual factors at the moment of interruption". borst2015makes also relate the length of interruption to the time of subsequent resumption and number of possible errors in a task.

3.2.2 Pausing behavior in CER

Pausing behavior has been studied in Computing Education Research (CER) both directly and indirectly in the context of computer programming. Similar to written language, where pauses between and within sentences are likely conditioned by different subprocesses [67], code writing has its own milestones and units of different level of complexity. When considering the mental effort needed to write code, one stream of research has focused on identifying and discussing plans and schemas for programming [69, 70]. It has been suggested that programmers who know the solution to a problem write their solution in a linear manner, while solving a new problem is done using means-ends analysis with the use of existing related schemas [69, 70]. Over time and through practice, accumulation and evolution of schemas allow programmers to solve problems more fluently, and also to learn to solve new problems with more ease [70, 71].

As discussed in Section 3.2.1, pauses can signify cognitive effort and are a natural part of the learning process. In programming however, an additional contributor to pauses, especially for novice programmers, are syntactic and semantic errors related to writing computer programs with the chosen programming language. These errors may be highlighted by the programming environment in use—similar to a word processing engine that shows spelling errors—as programming environments often highlight errors in program code, but they may be also visible through specific actions such as compiling the source code. These errors have been discussed especially in the context of Java programming, where researchers have studied the frequency of different types of errors [38, 56–59] and the amount of time that it takes to fix such errors [58, 72]. Denny et al. [58] and Altamdri and Brown [57], for example, have noticed that there are significant differences in the time that it takes to fix specific errors, and that over time students learn to avoid specific errors [57]. At the same time, the granularity of the data used in the analysis has an influence on the observed

errors [59] – different data granularity will lead to different observed syntax errors. In practice, collecting typing data with timestamps can provide more insight into the programming process over snapshot or submission data [18].

When considering syntax errors, pauses, and performance, the ability to fix syntax errors between successive compilation events has been linked with students' performance in programming [4], although it is unclear what the underlying factors that contribute to the observation are [73]. Including an estimate on the amount of time that individual students spend on fixing specific errors can increase the predictive power of such models [5, 20], highlighting the effect of time (or pause duration) on the learning process.

While the previous examples are specific to syntax errors and time, little effort has been invested into looking into pauses in programming. Perhaps the closest prior work to our work is that of Leppänen et al. [74] who studied students' pausing behavior in two courses, and identified that a larger quantity of short (1-2 second) pauses was positively correlated with course outcomes, while a larger quantity of longer (over 1 minute) pauses was negatively correlated with course outcomes. Our work builds on this by—in addition to correlation analysis—looking at pausing behavior over different contexts and also by investigating which characters precede pauses. Leppänen et al. hypothesized that one explanation for the correlation between long pauses and poorer course outcomes could be related to task switching between reading the course materials and solving the programming problems, but noted also that the pauses from writing code could be construed as instances of the student engaging in planning, reviewing, and translating the next ideas into code. Another possible hypothesis is related to differences in cognitive flexibility, i.e. the ability to fluently switch between two tasks; for example, Leinikka et al. [75] observed that students with better cognitive flexibility are faster at solving programming errors, although they did not observe links between cognitive flexibility and introductory programming course exam outcomes.

3.2.3 Typing and performance in programming

In CER, a multitude of data sources has been used for identification of factors and behaviors that contribute to course outcomes [76] – clicker-data [77,78], programming process snapshots [4–6, 9, 20], background questionnaires and survey data [79–83], and so on, but our focus is on keystroke data collected from programming environments [39, 84].

Keystroke data, or typing data, has been used, for example, for predicting academic performance [22, 24, 61], for detecting emotional states such as stress [60, 85], and for identifying possible plagiarism [55].

Much of the analyses of typing data that relate to students' performance has focused on between-character latencies, i.e. the time that it takes for the student to type two subsequent characters. This analysis has often focused on small latencies, as pauses have been considered as noise. For example, both Leinonen et al. [22] and Edwards et al. [24] used 750 milliseconds as an upper boundary for the between-character latencies. In general, these studies have found that faster typing correlates with previous programming experience and performance in the ongoing programming course.

Not all characters are equally important, however. For example, Leinonen et al. [22] identified differences in the time that moving from 'i' to '+' took for novices and more experienced students, while differences in some other character pairs were more subtle. Similarly, Thomas et al. [61] noted that the use of control functionality (e.g. using control and C keys) in general was slower than the use of e.g. alphanumeric keys, and the use of special keys such as delete and space was also slower than alphanumeric keys. Acknowledging that some of these latencies may be also influenced by the keyboard layout, they hypothesised that some of the latencies may be influenced by the thought processes related to the ongoing problem solving [61]. Our work builds on this prior work by examining which characters precede pauses, i.e. whether all characters are equally important when analyzing pausing behavior.

3.3 Methodology

3.3.1 Context and data

Our study was conducted in two separate contexts for purposes of generalization of the results.

University A

University A is a mid-sized public university in the Western United States. In a 2019 CS1 course, students used a custom, web-based Python IDE called *Phanon* [86] for their programming projects. *Phanon* logged keystrokes and compile/run events. Five programming projects, one per week, were assigned to the students during the study period. Each project consisted of two parts: a text-based mathematical or logical problem, such as writing an interest calculator; and a turtle graphics-based portion requiring students to draw a picture or animation, such as a snowman. A midterm exam was administered between the fourth and fifth project. There were three sections of the course all taught by the same instructor. Projects and instruction were the same for all three sections. In-person instruction was conducted three times per week. At the beginning of the semester students were given the opportunity to opt into the study according to our Institutional Review Board protocol, and this paper uses data only from students who opted in. The course was identical for students who chose to participate in the study and those who chose not to.

Gender information on participants was not collected, but in the course participants were recruited from, 19% self-identified as women and 81% self-identified as men. No information on previous programming experience, race or ethnicity was available for this study.

University B

University B is a research-oriented university in Northern Europe. The data for this study was collected from a 7-week introductory programming course in the Fall of 2016. The introductory programming course is given in Java and it covers the basics of programming, including procedural programming, object-oriented programming and functional programming. During each week of the course, there was a 2-hour lecture that introduced the core

Attribute	University A	University B
Instruction	Lectures w/sections	Lectures, sessions
Language (prog.)	Python	Java
Language (inst.)	English	Finnish
Participants	231	244
Prog. Environment	Web-based	Desktop

Table 3.1: Summary of contexts.

concepts of the week using live coding. The emphasis in live coding was in providing examples of how programming problems were solved with the concepts learned during the week, and in helping to create a mental model of an abstraction of the internals of the computer as programs are executed (introduction of variables, changing variable values, objects, call stack in a line-by-line fashion). In addition to the lectures, 25 hours of weekly support was available in reserved computer labs with teaching assistants and faculty.

The programming assignments in the course are completed using a desktop IDE accompanied with an automated assessment plugin [87] that provides students feedback as they are working on the course assignments. Combined with an automatic assessment server, the plugin also provides functionality for sending assignments for automatic assessment. In addition to the support and assessment, the plugin collects keystroke data from the students' working process, which allows fine-grained plagiarism detection [55] and makes it possible to provide more fine-grained feedback on students' progress. Students can opt out of the data collection if they wish to do so; the data collection was conducted according to the ethical protocols of the university.

Out of the 244 students at University B included in the study, approximately 40% self-identified as women and 60% as men. No information on previous programming experience nor race or ethnicity was available for this study.

Similar to the University A, University B had a midterm exam in the course. For the analyses conducted in this article, we focus on students' performance in the midterm examination. The contexts are summarized in Table 4.1.

3.3.2 Event and pause categories

Keystroke data was collected in both contexts. In the analysis, every keystroke of a student was categorized to an event. We consider eight event categories: (1) Alphanumeric keystroke, (2) Delete keystroke, (3) Return keystroke, (4) Spacebar keystroke, (5) Special character keystroke, (6) Tab keystroke, (7) Successful compile/run, and (8) Failed compile/run. Since the US context uses Python, in this paper we will call a compile/run event a "Run". The reasoning behind these categories is that they represent different tasks of the student: Alphanumeric events represent typing, Delete events indicate the student is preparing to make a correction, Run events represent a completion point where the student is ready to test the code, etc. The European context does not have information on tabs or the status of run events, so analyses relating to the status of run events or tabs will only use data from the US context.

For the analysis of pauses, we chose to use four types of pauses. While pausing analyses in the context of programming have been exploratory [74], research in pausing in language production varies in terms of pause thresholds. A lower bound of 1-2 seconds appears to be the most common [88–90], and thus, we adopted a 2-second lower bound for our study. Taking into account research on working (short-term) memory time capacity [91], a meaningful upper bound is at 15 seconds – pauses between 2 and 15 seconds may reflect thinking about the code on a low, "local" level, including thinking about the syntax, and could be tied to working memory. We call these pauses *micro* pauses.

Short pauses may reflect higher-level processing like planning the following code segment, setting the next sub-goal, and revising code similar to the production stage of written language or some kind of distraction. We chose 180 seconds to be the upper bound for *short* pauses.

Mid pauses, up to 10 minutes, may reflect voluntary or problem-solving related breaks. We hypothesise that students who have difficulties may consult learning materials or visit other resources in search for help or for refreshing their memory. Such a continuous pause would cause longer task resumption [92]. Finally, *long* pauses, greater than 10 minutes, are

most likely to stand for task disengagement as noted by prior work [93]. We expect such pauses to take place after finished code segments or compilation.

For simplicity, we also refer to pauses initiated by a certain event type using the name of the event type. For example, a *delete* pause is a pause initiated by the student pressing the delete key; the last event before a *failed run* pause is a *failed run* event. Pauses preceded by other event types are named similarly.

Because of data availability, we use a single measure of outcome – exam score. In the US context we use the score of a exam that falls just before the last project in the study. In the European context, we use the exam score from the first out of three programming exams. The first exam was organized on the third week of the seven-week course.

3.3.3 Statistical tests

We report p values of all statistical significance tests, of which there are 95. We follow the American Statistical Association's recommendations to use p values as one piece of evidence of significance, to be used in context [94], though we do suggest $p < 0.05/95 \approx 0.0005 = 5e^{-4}$ as a reasonable guideline for credible p values [95]. When considering the claims in our work, we suggest taking into account the additional supports beyond single p-values. For example, claiming that *delete* pauses are negatively correlated with exam score is based on consistent negative correlation across pause types in both studied contexts. For distribution comparisons we use the t-test (as the data appears normal) with Cohen's d effect sizes and for correlation we use the Pearson r statistic.

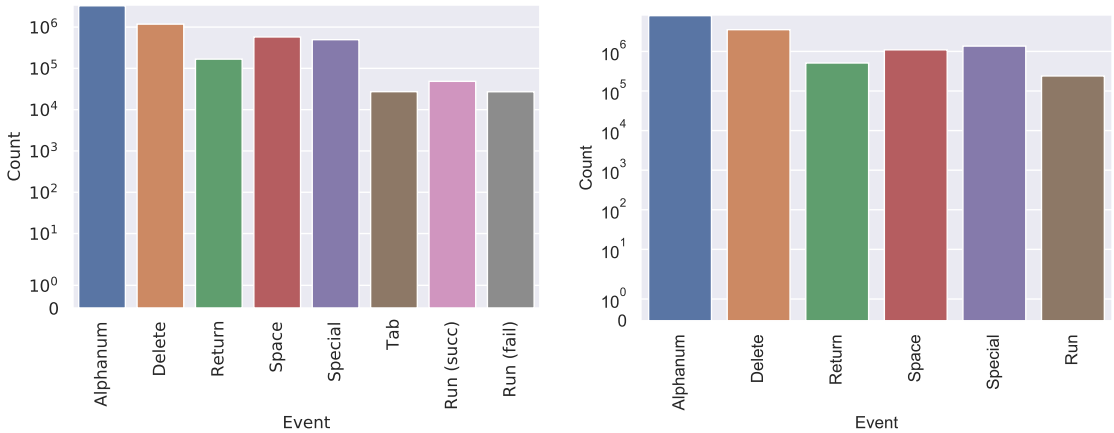
3.4 Results

3.4.1 Descriptive statistics

Table 3.2 shows descriptive statistics of our study and Figure 3.1 shows the distribution of event types for each of the two contexts. *Alphanumeric* keystrokes are the most common event, with *space* (spacebar) and *special characters* also being common. *Run* events and the *tab* keystroke are less common. Both contexts use an editor that automatically indents

Context	Students	events/ student	pauses/ student
Python (US)	231	25186 ± 11243	2183 ± 1009
Java (Europe)	244	54698 ± 25538	6774 ± 3189

Table 3.2: Descriptive statistics of the study.



(a) US/Python (b) European/Java

Fig. 3.1: Log-scale bar chart showing the total number of events for each type. a Number of events in the US/Python context. b Number of events in the European/Java context.

the next line of code after a *return* key press, which likely contributes to the observed lack of tab keystrokes.

A difference between the contexts is the relative frequency of *run* events – students in the Java context run their code far more often than those in the Python context. This is likely not due to the language, but the way the courses are organized. In the Python context, students had one large assignment due each week, while Java students had tens of smaller assignments due each week. We conjecture that the smaller assignments induced the students to run/compile their code more often.

3.4.2 Frequency of pauses

We calculated a measure of pause frequency as $\frac{p_l}{n}$ where p_l is the number of pauses of length l and n is the total number of events. In the Python context, on average, student pause frequency is 0.09 ± 0.02, meaning, on average, students execute 11 events before

pausing for two seconds or more. Most pauses are *micro* pauses, which have a frequency of 0.07, followed by *short*, *mid*, and *long* pauses with frequencies of 0.02, 0.001, and 0.001, respectively.

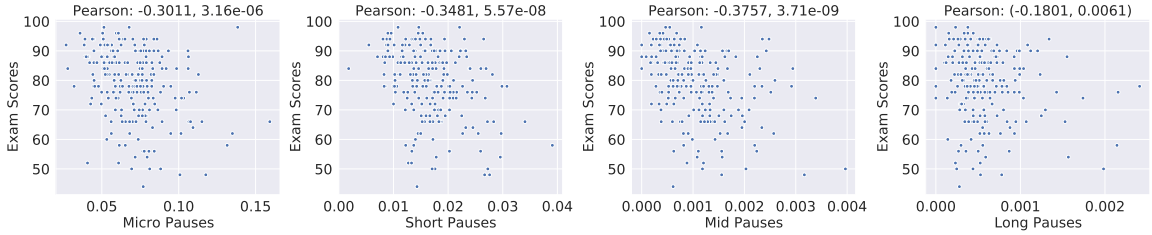
The Java context was somewhat different: on average, student pause frequency is 0.13 ± 0.03, meaning, on average, students execute 8 events before pausing for two seconds or more. Most pauses are *micro* pauses, which have a frequency of 0.09, followed by *short*, *mid*, and *long* pauses with frequencies of 0.03, 0.002, and 0.001, respectively.

As might be expected, Figure 3.2 shows a negative correlation between pause frequency and exam score, meaning students who are pausing more often are performing worse on the exams. In the Python context, this correlation is consistent across *micro* ($r = -0.30, p =$ 3.16e-6), *short* ($r = -0.35, p =$ 5.57e-8), and *mid* ($r = -0.38, p =$ 3.71e-9) pause lengths, with a weaker correlation for *long* ($r = -0.18, p = 0.0061$) pauses. The Java context, in contrast, has a weaker correlation for the *micro* pause ($r = -0.11, p = 0.0654$) than for the *short* ($r = -0.20, p = 0.0013$), *mid* ($r = -0.23, p = 0.0003$), or *long* ($r = -0.22, p = 0.0006$) pauses. In general, the correlations for the Python context are stronger than for the Java context, although as seen in Figure 3.2, the Java context has a noticeable ceiling effect in the exam.

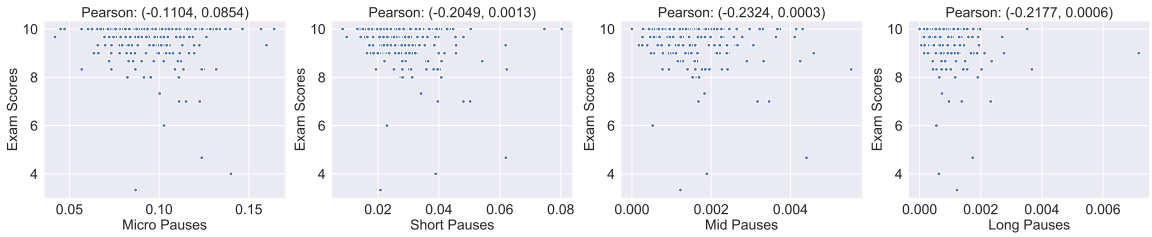
Figure 3.3 shows correlations between the number of different type of pauses that students take, i.e., whether students who are pausing for short amounts of time are also taking longer pauses. All types of pauses are at least moderately correlated with all other types of pauses (see Figure 3.4). Interestingly, the correlations weaken as the pause lengths grow for the Python context, while the Java context shows a strong correlation for the *mid*/*long* pause pair.

3.4.3 Student types

To characterize students, we represent each student using a vector that contains the relative proportions of each pause type. For example, a student represented with a vector $[0.80, 0.15, 0.03, 0.02]$ has 80% *micro* pauses, 15% *short* pauses, 3% *mid* pauses, and 2% *long* pauses. Since the vector is a partition of unity, the feature vector has only three degrees of

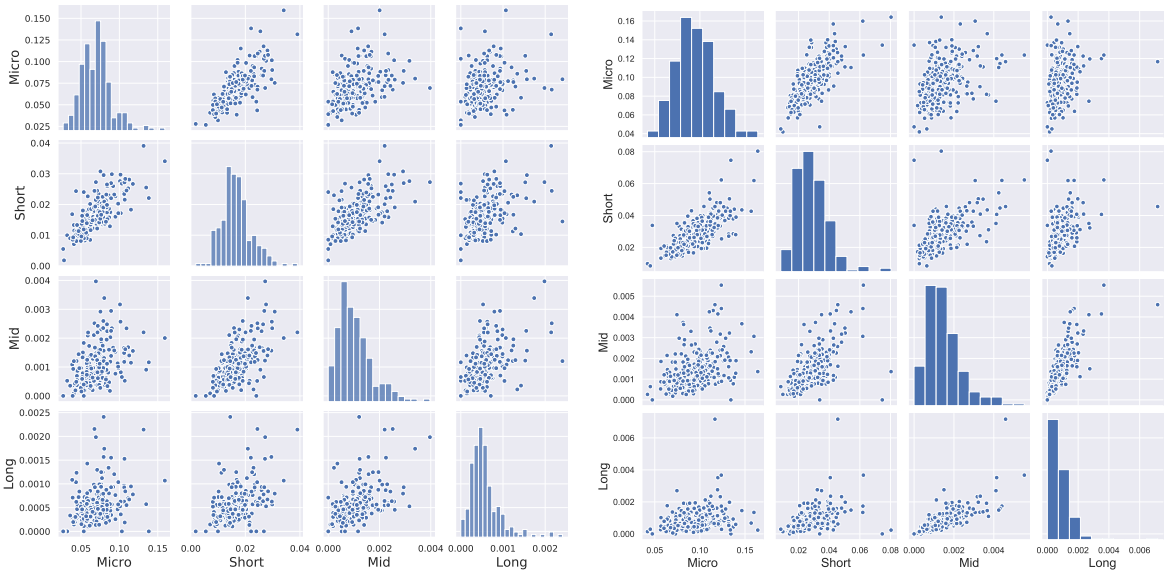


(a) US/Python



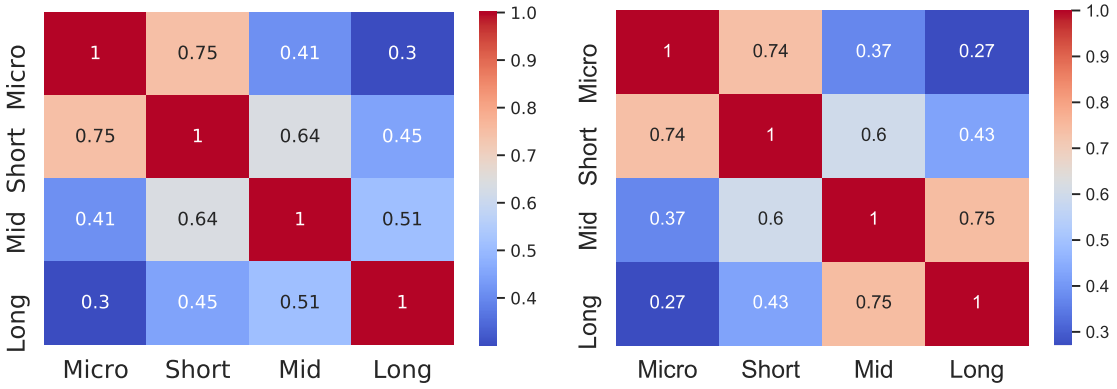
(b) European/Java

Fig. 3.2: Frequency of the different type of pause correlated with exam score. Frequency is calculated as number of pauses divided by total number of events.



(a) US/Python (b) European/Java

Fig. 3.3: Correlations of total pauses with each other per student. As expected, the number of micro pauses a student takes has a strong positive correlation with the number of short pauses. While there are still strong and medium correlations between shorter and longer pauses, the correlations become weaker.



(a) US/Python (b) European/Java

Fig. 3.4: Similar to Fig. 3.3, this figure shows correlation coefficients for the different pause lengths.

freedom, though, for clarity, we represent it here with all four coordinates.

To identify student types based on the vector representations, we use k-means clustering to cluster students into student types. Using the elbow method (visually finding the "elbow" of a line chart of number of clusters against explained variance [96]) to identify a good number of clusters, we chose $k = 2$ for interpretability, though, as we will see in Section 3.5.2, the choice of k is not particularly important in this case.

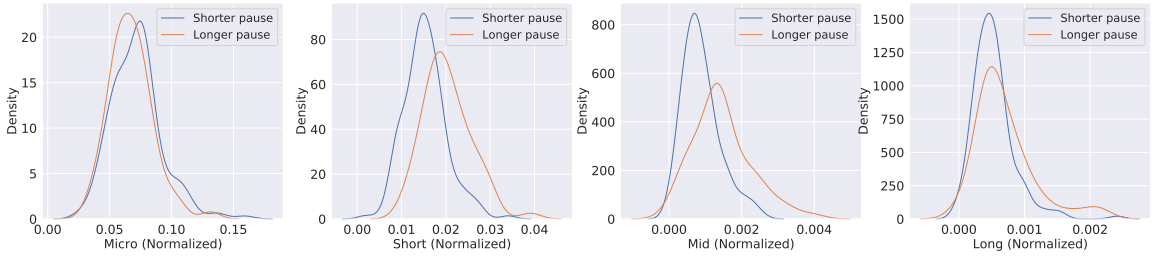
We see in Table 4.2 and Figure 3.5 that one group of students in each context took relatively more *short*, *mid*, and *long* pauses than the other group, although in the Java context, the difference is less pronounced. We call the clusters the *longer pause* and *shorter pause* groups, respectively. When examining the groups and exam scores, we observe that the students in the *shorter pause* group had higher exam scores than those students who took longer pauses. The distributions of pause frequencies are approximately normal and t-tests suggest that there is a difference between *short*, *mid*, and *long* distributions.

3.4.4 Initiating pauses

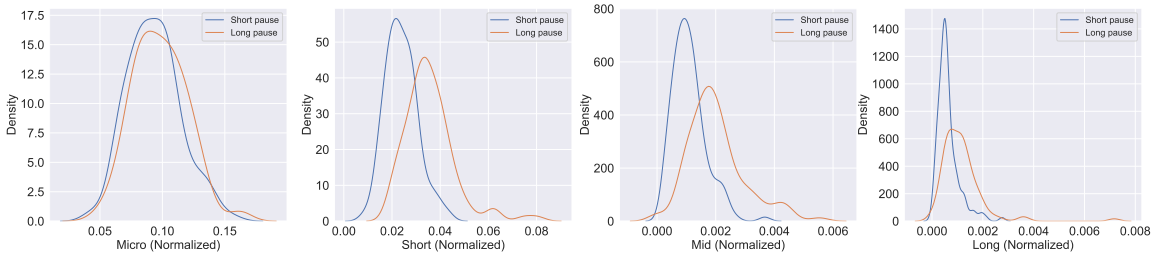
In Figure 3.6 we see relative frequencies of event types by pause length. We define relative frequency for an event type E as the percentage of pauses of a given length initiated by an event of type E. For example, in Figure 3.6 we see that in the Python context,

Context	Cluster	Centroid Micro	Short	Mid	Long	Students	Average keystrokes $\times 10^4$	Average pauses $\times 10^3$	Average exam
Python (US)	*shorter*	0.81	0.17	0.010	0.0060	71% (164)	2.6 ± 1.1	2.3 ± 1.0	80.2 ± 11.3
Python (US)	*longer*	0.75	0.23	0.016	0.0077	29% (67)	2.4 ± 1.3	2.1 ± 1.1	76.2 ± 11.5
Java (Europe)	*shorter*	0.79	0.20	0.01	0.01	57% (139)	5.2 ± 2.1	6.2 ± 2.8	9.59 ± 0.90
Java (Europe)	*longer*	0.72	0.26	0.01	0.01	43% (105)	5.8 ± 3.0	7.6 ± 3.5	9.35 ± 0.88

Table 3.3: Statistics of the clusters for the two contexts, US and European. For the Cluster column, "*shorter*" means "*shorter pause*" and similar with *longer*. A t-test for the two distributions of exam scores yields $(t = 2.2, p = 0.026, d = 0.35)$ in the US context and $(t = 2.1, p = 0.034, d = 0.28)$ in the European context.



(a) US/Python



(b) European/Java

Fig. 3.5: Clustering. In the Python context, t-test statistics (t, p) and effect sizes (d) between the two distributions are: Micro $(t = 1.5, p = 0.12, d = 0.23)$, Short $(t = -6.9, p = 3.6e{-}11, d = -1.01)$, Mid $(t = -6.4, p = 8.4e{-}10, d = -0.93)$, and Long $(t = -3.2, p = 0.0014, d = -0.47)$. In the Java context, t-test statistics (t, p) and effect sizes (d) between the two distributions are: Micro $(t = 1.7, p = 0.08, d = -0.22)$, Short $(t = -10.8, p = 2.4e{-}22, d = -1.35)$, Mid $(t = -9.3, p = 1.1e{-}17, d = -1.15)$, and Long $(t = 2.1, p = 2.6e{-}9, d = -0.77)$.

alphanumeric keystrokes initiate 27% of all *micro* pauses (2-15 seconds) and 15% of *long* pauses (> 10 minutes) while accounting for 57% of all events, regardless of whether the events initiated a pause or not. In the Java context, the distribution related to *alphanumeric* events that start a pause is very similar. Roughly 29% of *micro* pauses and 18% of *long* pauses are initiated by the events while they account for 28% of all events.

Certain types of events in both contexts decrease in frequency with increasing pause length. *Alphanumeric*, *return*, *space* and *special characters* seem to follow this trend preceding to a greater extent shorter pauses.

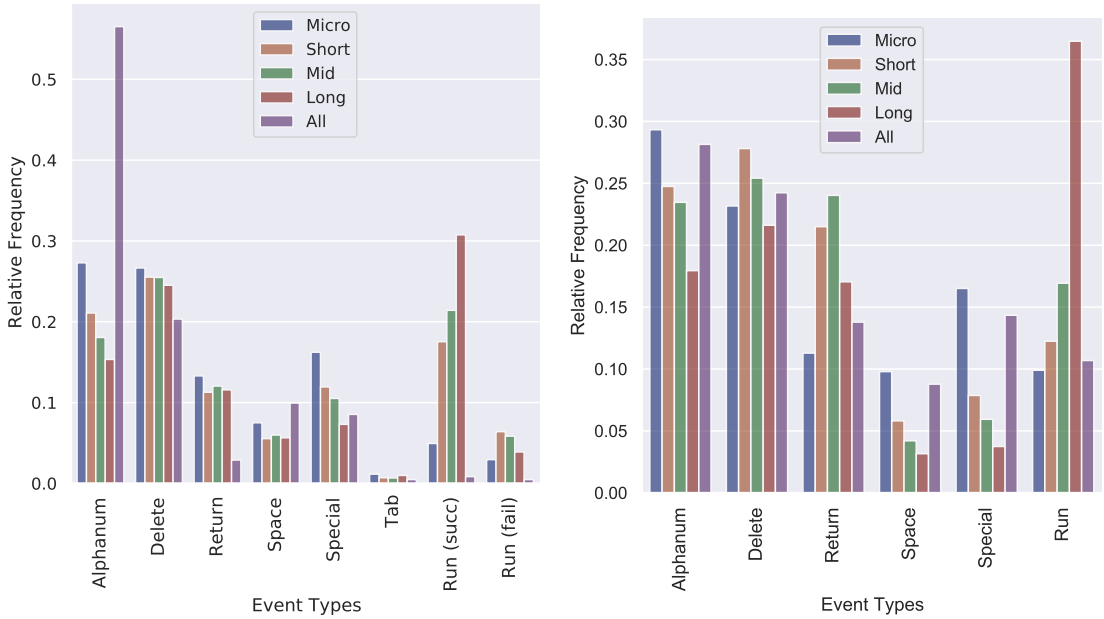
In Table 3.4 we see that *alphanumeric* events initializing *micro* pauses have a positive correlation with exam score, but that the correlation weakens until it is not detectable for *long* pauses. Conversely, pausing after *special characters* is not necessarily correlated with success. In fact, a weak negative correlation exists with *special characters* initializing *micro* pauses.

In the Python context, the percentage of pauses preceded by the *delete*, *return*, and *space* keystroke events remains roughly the same across pause lengths (Figure 3.6). The *return* keystroke is unique among the three in that, despite being so infrequent in the data, it precedes so many pauses (11-13%). This tendency does not repeat in case of *delete* and *space* events.

In the Java context, the situation is different. *Return* and *space* events show steady decline in percentages of preceding pauses. The longer the pause, the less common it is for those event to precede it. The opposite applies to the *delete* events. This could be accounted for differences in programming languages and their relations to students' native languages [27]. Even though the deleting behaviour differs across the contexts, correlation of most *delete* pauses with exam scores remains negative in both cases.

Both *successful* and *failed run* attempts have disproportionate prominence among events preceding pauses relative to their overall frequency.

3.5 Discussion



(a) US/Python

(b) European/Java

Fig. 3.6: Grouped bar chart showing normalized/relative frequencies of keystrokes by pause length. "All" are all events, whether they precede a pause or not.

	Pause length	Enter		Alphanum		Delete		Special		Space		Tab		(Success) run		Fail run	
		r	*p*	*r*	*p*	*r*	*p*	*r*	*p*	*r*	*p*	*r*	*p*	*r*	*p*	*r*	*p*
Python	Micro	0.27	1e−5	0.28	1e−5	−0.33	1e−7	−0.23	4e−4	0.21	0.001	0.058	0.41	−0.17	0.015	−0.38	1e−7
	Short	0.11	0.1	0.19	0.004	−0.245	1e−4	0.01	0.81	0.19	3e−3	0.062	0.42	−0.05	0.45	−0.38	1e−7
	Mid	0.076	0.29	0.14	0.04	−0.13	0.05	0.13	0.07	0.14	0.08	0.22	0.23	−0.007	0.91	−0.22	8e−4
	Long	−0.022	0.80	0.005	0.95	−0.12	0.11	−0.03	0.74	0.037	0.73	0.13	0.57	0.067	0.32	−0.06	0.37
	All	0.2687	3e−5	0.29	5e−6	−0.36	1e−6	−0.20	0.002	0.22	5e−4	0.073	0.30	−0.13	0.04	−0.43	1e−6
Java	Micro	0.18	4e−3	0.31	0.0	−0.20	9e−4	−0.18	3e−3	−0.029	0.64			−0.034	0.58		
	Short	−0.012	0.84	0.21	5e−4	−0.16	0.010	0.055	0.36	0.021	0.73			−0.024	0.69		
	Mid	-0.028	0.66	0.23	2e−4	−0.23	2e−4	0.023	0.72	0.034	0.60			0.079	0.19		
	Long	−0.012	0.86	0.033	0.62	0.067	0.28	−0.24	2e−4	0.014	0.86			0.070	0.25		
	All	0.082	0.18	0.23	1e−4	−0.24	1e−4	0.21	5e−4	0.12	0.042			−0.088	0.15		

Table 3.4: Pearson r correlations with p values between a student's tendency to initiate a given length of pause with a given event type and exam score. "All" indicates percentage across all events (both those initiating pauses and not). We do not have data on tabs or whether a run was successful or not in the Java context, so tab values are not included for the Java context and the *(Success) run* column should be interpreted as a successful run for the Python context and all runs for the Java context.

3.5.1 Frequency of pauses

Before answering the first research question, *Is there a correlation between the relative number of pauses a student takes and their performance (exam score)?*, we checked whether our bucketing was sensible by performing a correlation test. As we can see from Figs. 3.3 and 3.4, there are correlations between all types of pauses which is not surprising since the pauses lengths are on the time continuum. However, neighbouring types of pauses do not show a very high degree of similarity, which justifies our choice. Moreover, *micro* and *short* pauses, having the highest correlation coefficient, yield quite different correlation coefficients in terms of relationships with exam scores (see Figure 3.2).

In general, we observe that students who pause more often perform more poorly on exams (Figure 3.2), which is in line with the results observed by Leppänen et al. [74]. This effect is not large, but it is consistent across pause types and contexts. We note that this measurement is frequency of pauses, so it is normalized across students regardless of the number of total events they execute. In this paper we do not make any claims regarding what students were doing during their pauses, whether they were thinking, drawing on other resources, or disengaged. But the correlations in our data indicate that regardless of pause activity, pauses correlate negatively with exam score, at least in the aggregate. We note that certain activities may not cause negative correlation with achievement, but it appears that these activities are in the minority and are dominated by negative-effect activities.

Frequency of *mid* pauses (3-10 minutes) in both contexts have the strongest negative correlation with exam score of all the pause types. Comparing to the *long* pause which does not have an upper bound, it is clear that after at most 10-minutes long *mid* pause students get back to typing. We conjecture that *mid* pause may be the most harmful because it potentially can cause the longest resumption. If the activity taking place during the pause is not related to the task, the pause may be treated as irrelevant interruption [97]. According to altmann2007timecourse and many others (for example, see [98–100]), the length of interruption correlates with the time of task resumption and numbers of possible errors.

3.5.2 Student types

Our second research question is: *What groups of students exist when clustering on pausing behavior?* We clustered students into two types, *longer pause* students and *shorter pause* students. *Shorter pause* group tends to take proportionally more *micro* pauses, whereas *longer pause* students take fewer *micro* but more of *short*, *mid*, and *long* pauses. The *shorter pause* students appear to perform better in the exam in the both contexts.

In a sense, grouping students into clusters is arbitrary: Figure 3.2 shows that pauses of all lengths are negatively correlated with exam score, indicating that the 4-dimensional feature vectors are not linearly independent, effectively making our clustering single-dimensional and not particularly interesting regardless of choice of k. Nevertheless, the analysis reveals one difference between the contexts that may be of interest: in Table 4.2, we see that more events correspond to more pauses across the contexts. However, groups which produce more events are not the same. In the US/Python context, the *shorter pause* group tends to type and pause more, whereas in the European/Java context the opposite applies. Additionally, proportions of pauses in the US/Python context remain roughly consistent across the groups and equal to 0.09 and in the European/Java context similarly, being 0.12 and 0.13. This observation could be due to a number of context-specific factors, such as the way how each context uses programming assignments.

3.5.3 How pauses are initiated

Our third research question is: *What events initiate a pause and how does this correlate with the performance of the student?* The first thing to note is that the distributions of event types, for each of the four pause lengths, do not match the overall distribution of events (Figure 3.6). This confirms, as one might expect, that, in general, students are not pausing at arbitrary times, meaning that pauses are generally purposeful and not taken at random times while typing.

Deletes and *failed runs*

There is some abruptness regarding what initiates a *long* pause. *Long* pauses, those

of 10 minutes or more, may indicate that the student is disengaged from working on the project [93]. One would expect the most natural way to take a break would be a *successful run*. Yet, in the Python context, only 30% of *long* pauses are initiated as such. Another intuitive, natural break would be a *failed run*, as the student might need a break or an extended session of reviewing external materials after a failure. Yet *failed runs* account for only 4% of *long* pauses. This means that 66% of long pauses are initiated with a keystroke. The most common event for *long* pauses, the *delete* keystroke, initiates 25% of the pauses. The Java context is similar, with 22% of *long* pauses initiated by *delete*. This seems remarkable. A *delete* press often indicates an error and so, after the *delete* press, the student needs to execute keystrokes to replace the incorrect code. At times, however, students are taking a break instead of completing the correction. If this happens it could indicate that the student may lack motivation, diligence, or the corrective know-how without consulting external help. Other types of pauses were also rather often preceded by a *delete* event (26-27%). From Table 3.4, we can see that the correlations of the exam score with such pauses are negative. This may signify that *deletes* are used less often for removing unneeded code (e.g., print statements or comments) and more often when students are confused and do not know how to proceed. This same reasoning could be used to explain the negative correlation of *failed runs* initiating pauses, i.e., that an extended pause after a *failed run* indicates the student does not know how to fix the problem and has to take time to either consult other materials or take a break. Indeed, the correlations of *failed runs* with exam score closely mirror those of *delete* key presses.

One could suggest that the consistent negative correlations of *delete* and *failed run* events initiating pauses with exam score simply reflect the overall correlation of these event types with exam score. We note, however, that distributions of the two events are different and they demonstrate different degree of involvement in a *long* pause initiation. While *deletes*, constituting 20%/24% of all events and preceding 25%/22% of *long* pauses, *failed runs* account for only 0.5% of all events but precede 4% of long pauses. Thereby, it is six times more likely that *failed run* event will initiate a long pause than a delete keystroke. A

plausible interpretation of this observation is that students are deliberately pausing after failed runs, at least more often than after deletes.

Events decreasing in frequency with pause length

Alphanumeric keystrokes are what we might call "middle" events – they are the most common while being somewhat less significant in terms of reflecting thinking processes. The fact that the frequency of *alphanumeric* events preceding a pause decreases with increasing pause length as much as it does (Figure 3.6) suggests that students are completing their lower-level processing thoughts before taking longer breaks. Indeed, it appears that students are deliberate in taking longer breaks rather than getting interrupted, as would be the case if *alphanumeric* pauses were more common.

In Python, statements generally do not end with a *special character* as they do in Java (e.g. semicolon for a single-line statement and closing brace for a block). So it is not surprising that pauses initiated by *special characters* decrease in frequency with increasing pause length in the Python context. What is surprising, however, is that the Java context has a very similar phenomenon. We expected longer pauses to be frequently initiated by *special characters* in the Java context, as ending a line with a semicolon seems like a natural stopping point. We do not know why this is not the case, but we suspect that this is, again, a consequence of the difference in instructional methods between the two contexts. The Java students work on smaller projects and run more often, and so they may be more likely to complete their thought or work session with a *run* event.

In Table 3.4 we see that, in both the Python and Java contexts, *alphanumeric micro* pauses are positively correlated with exam score while *special character micro* pauses are negatively correlated. As these two event types behave similarly in other respects, we discuss a possible explanation for this difference. Roughly half of *special characters* require further processing: an open parenthesis expects formal parameters for a function call; quotes expect a string; an open bracket expects list/array indices; etc. It may be that a *micro* pause, which may last as many as 15 seconds, indicates student hesitancy and lack of fluency with Python or Java syntax. This lack of fluency with a fundamental aspect of programming may be

why the student exam scores are lower. Problems with special characters being indicative of struggling has been hypothesized also in previous work [22, 24]. If *special character* pauses do indicate an uncertainty with syntax then instructors may consider an increased focus on syntax fluency for students initiating pauses with *special characters*.

Events constant across pause length

In the Python context, the percentage of pauses initiated by the *return* event remains roughly the same across pause lengths (Figure 3.6). This makes sense in the context of both shorter and longer pauses: pressing *return* requires short-term planning for the next line, so its prevalence among *micro* and *short* pauses is logical; pressing *return* is also a natural stopping point before taking a break, so it is frequent among mid and long pauses. Being every 33rd event in the Python data and every 7th in Java data, return initiates approximately 12% of any type pauses in the Python context and as much as 12-24% of of any type pauses in the Java context. This seems to confirm the return keypress being a natural stopping point.

Run events

Being rather rare in the typing data in both contexts, *run* events are notably evident among events preceding pauses, especially *short*, *mid* and *long*. This is not unexpected: it would be highly unusual for a *run* event to take fewer than two seconds, so the great majority of *run* events would precede at least a *micro* pause. A large proportion of *long* pauses are initiated by *successful runs* in the Python context and *runs* in the Java context. It is instructive to consider why students would pause after a *successful run*. It is possible that a student takes a pause to consult external resources (e.g., internet, textbook, another person) regarding how to proceed with their program, but it seems more likely that the student would have at least an idea of what to do next after a *successful run*. Therefore, we suggest that the more likely scenario is that the student is instead disengaging from working on their assignment. If this is the case, then we could possibly use the percentage of *successful run* long pauses in the Python context as a lower bound for the number of

long pauses in which students are disengaging. In our data, this indicates that students are disengaging during at least 30% (roughly) of *long* pauses. We expect that this is a conservative lower bound.

3.5.4 Threats to validity

Internal validity: As is natural in educational studies, our study comes with an inherent self-selection bias. It is possible that the way the studied courses were organized and the way the student population at both universities is formed influences the observed outcomes. It is unclear, for example, whether similar results would be observed if the study would have been conducted in the context of primary or secondary education, or in life-long learning. When considering the outcome of the courses, we used exam score as a proxy for performance, which can be affected by factors such as exam stress. In addition, the European/Java context had a noticeable ceiling effect in the exam outcomes. It is possible that this also influenced some of our findings and that lifting the ceiling effect would affect the correlations.

External validity: We studied keystrokes in two contexts to increase the degree to which our findings can be generalized to other contexts (see Section 3.3.1). The strength of the correlations and the p values varied somewhat between the contexts and we cannot state which context-specific factors contributed to the differences.

3.6 Conclusions

In this article, we presented an analysis of keystrokes with an eye toward understanding pausing behavior of CS1 students and its implications on academic outcomes. In this section we draw conclusions from our results in each of our three research questions.

RQ1 *Is there a correlation between the relative number of pauses a student takes and their performance (exam score)?* We observe that negative correlations between pause frequency and exam score exist as illustrated in Fig. 3.2. The most illustrative is the frequency of *mid* pauses – those of length 3-10 minutes. We suggest that these pauses indicate that a student may be distracted easily, but it could also indicate students who are spending time

using external resources for help on their projects. Revesz2017 suggests, since keystroke logs alone do not allow us to "make inferences about the specific cognitive processes that underlie pausing behaviors", that combining event logs with "other techniques such as verbal reports and eyetracking" could be helpful in obtaining more detailed information. Further study could help us understand what these students are doing during pauses and what they were working on when they paused. But in the meantime, the pause/exam score correlation appears actionable. We suggest that a tool that allows practitioners to visualize students' pausing behavior could be particularly useful. In addition, as previous studies that have used keystroke data for predicting course outcomes have mainly focused on latencies smaller than 750ms [22,24], future research should seek to combine such keystroke data with pausing data and study whether these phenomena have the same underlying tacit factors.

RQ2 *What groups of students exist when clustering on pausing behavior?* We found in a cluster analysis that students whose pausing behavior tended toward short pauses performed better in general on exams. The cluster analysis primarily indicated a correlation between typical pause length for a student and exam score. When considering the identified student types in the light of CER studies that have identified student types such as the tinkerers, stoppers, and movers [56,101], most of the students in the studied contexts could be categorized as movers, despite the differences in their pausing behavior. As pausing is linked with cognition and thought processes, and as writing code is linked with a multitude of factors including understanding syntax and the given problem [102,103], further research is needed to understand the lack of stopping and the differences in pausing.

RQ3 *What events initiate a pause and how does this correlate with the performance of the student?* We have presented evidence that pauses do not occur randomly while a student is programming – students tend to finish their thoughts and pause after a natural stopping point. This observation is in line with the studies on student cognition and programming and how students solve programming problems [69, 70], where students write constructs informed by schemas that engage procedural memory. Fully 25% (Python) and 22% (Java) of *long* pauses (>10 minutes) are initiated by *delete* events. We suggest that students who

pause after *delete* are possibly less engaged (taking a break instead of writing the code to replace the deleted characters) or they lack the knowledge to write a fix (consulting external resources to learn how to fix the problem). This presents interesting questions for future research, such as what percentage of *delete* pauses indicate a disengaged student. Beyond identification of at-risk students, the negative correlations of *special character* and *failed compile* pauses suggest possible pedagogical and material innovations to improve student fluency after special characters and minimize the number of failed runs.

In addition to the directions for future research discussed above, there are additional avenues for further research. As an example, while previous research in syntax errors has noted that there are differences in the time that it takes to fix syntax errors [57,58], our study highlights that pause durations are related to the pressed keys. Combining information on present syntax errors (or the lack of them) with information on pauses could create more in-depth understanding of students knowledge and actions – for example, pauses preceded by a syntax error likely indicates different thought processes than pauses not preceded by a syntax error. Similarly, looking at what syntactic construct was just typed or is being typed could affect pausing behavior. While our definitions of pauses were based on related literature (e.g., [74]), future work could explore alternative bins, including higher resolution bins for the micro pause, which spans lengths from 2 to 15 seconds in the work reported in this paper. Language specific differences should also be studied further – as an example, we noted that in the European/Java context students took a *micro* pause on average after 8 keystrokes, while students in the US/Python context took a *micro* pause on average after 11 keystrokes. It would be meaningful to understand where this difference stems from. If it is simply the language, then one possible implication is that the relative verbosity of Java when compared to Python would not only require the students to type more, but also to pause to think more. On the other hand, if it is a product of a contextual factor, then it could be something that could be sought to disseminate to other contexts as well. Future studies could also focus on differences between the beginning and end of the course to see if programming behavior changes with experience.

CHAPTER 4

Circadian Rhythms of CS1 Students

4.1 Introduction

Circadian rhythms – the cycles of our internal clock – influence our daily activity and productivity. Achievements in academic studies [104], programming [105], and other domains are affected by circadian rhythms. Chronotype, or diurnal preference, is a person's tendency toward activity at certain times of day and is thought to be a natural characteristic of an individual [106].

Gaining an understanding of students' internal clocks and time-management behaviors has the potential to inform individualized recommendations for learning as well as targeted interventions for helping those struggling with managing their time. With that goal, we study introductory programming students from two contexts, one in the US and one in Europe, with different geographical location leading to differences (e.g. available daylight). In addition, the contexts have different teaching approaches and use different programming languages. In both contexts, we collected timestamped keystroke data from students' programming process for the purposes of gaining a deeper understanding of students' behavior. Prior work in computing education has used keystroke data, for example, to predict students' success [17, 22, 24] and to identify students taking an exam [11, 23]. Our overarching objective is to study the students' possible chronotypes and preferences, as evidenced by students' behavior inferred from the timestamped keystrokes, and look for connections of these chronotypes and preferences with assessment outcomes.

In the context of software engineering education and computing education research, there exist streams of research on students' time management practices and self-regulation [74, 107, 108]. Much of this has had a focus on behavior related to deadlines, including procrastination, and ways to increase the earliness of students' work [108–110]. In general, these

studies agree that starting early leads to better learning outcomes, but little focus has been put on at what times students typically work and how these times contribute to students' learning outcomes, despite some existing research on software developers' working hours and bugs introduced in code commits made at different times during the day [105, 111] .

Our work offers three contributions. 1) We use unsupervised machine learning to identify chronotypes among introductory computer programming (CS1) students and find that these chronotypes match up remarkably well with chronotypes of general populations reported in the literature. 2) We find strong linkages between chronotype and academic outcomes, again, consistent with the literature. We also highlight circadian rhythms as a viable, important, and relevant topic for both the computing education research community as well as for computing educators.

This article is organized as follows. Next, in Section 4.2, we outline the theory and related work upon which our article builds. Section 4.3 describes the contexts in which the study took place and outlines the research questions and methodology. Results are presented in Section 4.4, and discussed in Section 4.5. Finally, conclusion and potential streams for future work are outlined in Section 4.6.

4.2 Background and Related Work

4.2.1 Circadian Rhythms and Chronotypes

A *circadian rhythm* is a cycle of internal oscillations in nearly all physiological activities generated by the molecular circadian clock and has a period of approximately 24 hours [112, 113]. *Chronotype* is a person's natural inclination toward activity at certain periods of the day and depends on a circadian rhythm for synchronization [106]. Research suggests that the circadian cycle is conditioned by a group of clock genes [114], which explains individual differences. Nevertheless, it is not fixed and it does shift during an individual's lifetime.

People have to fit in daily activity according to their schedules taking into consideration social constraints (e.g., typical work times, services work hours, etc.). That is, clock time

preferences are more likely to match a chronotype when an individual has more flexibility in activity organisation. Discrepancy between an individual's chronotype and schedules determined by social constraints causes a phenomenon called *social jetlag* [106, 115]. It leads to accumulation of a "sleep debt" with subsequent feelings of tiredness and drop in cognitive abilities throughout a working week. Research [116, 117] shows that sleep-wake times of people whose weekday routine matches their chronotype do not change during weekends contrary to the case of people whose timing inclination of natural activity contrast with their weekday routine. Since both activity and sleep comply with chronotype in natural conditions, there is a premise to consider weekends as a reflection of true chronotypes.

Most studies distinguish between two chronotypes: *morning* and *evening*. In recent years a third type – *intermediate* has emerged (for example, in [118, 119]). In 2019, Putilov et al. [120] formed an argument that there are four chronotypes which can be distinguished by varying times of sleepiness and alertness: *morning*, *afternoon*, *napper* and *evening*. The ***morning*** type prefers to wake up early in the morning and is most alert from 9 a.m. to 11 a.m., after which the alertness curve gradually goes down. The ***napper*** type has two peaks of activity – the first from 9 a.m. to 11 a.m., and the second from 3 p.m. to 10 p.m. The ***afternoon*** type is alert from 11 a.m. to 6 p.m., gradually decreasing until 9 p.m. Finally, the ***evening*** type, similarly to the napper type, has two peaks, although the first one is less evident and falls into period of 11 a.m. to 2 p.m., while the second period of increased alertness is from 6 p.m. until late night. We use Putilov's et al. chronotype classification in our study.

Relationships between chronotype and academic achievement of adolescents have been discussed across countries (e.g., [118, 121–124]). There exist consistent patterns of negative correlation between "eveningness" and indicators of academic performance (e.g. GPA, exam grades), whereas "morningness" is positively correlated with academic achievement. It is possible that these correlations are at least partly due to the *synchrony effect* [125], where an individual's cognitive performance is optimal during certain times of day and suboptimal at other times. That is, a student working during hours that are suboptimal for them because

of socially driven schooltime hours [106, 115] can lead to sub-par academic achievement.

Attention, working memory, and executive functions are affected by the synchrony effect [126–128], as well as the ability to inhibit distractors and focus on a specific task [129]. Furthermore, the synchrony effect is more pronounced when tasks are more difficult [130, 131].

4.2.2 Chronotypes in Software Engineering

Recent research by Claes2018 examined software developers' code contributions to a large number of open source projects. The study showed that most (2/3) of the developers followed typical working hours. A small cluster of developers tended to work outside the regular daily schedule and also contributed on weekends. Although those who worked late hours and weekends could be explained by the "evening" chronotype of the developers, an alternative explanation is that, since the study used open source projects many of which pay little or no salary to contributors, developers were working another job during normal working hours. The times of code contributions have also been linked with occurrence of bugs [105], providing potential evidence of sleep deprivation and inconsistency with individual circadian rhythms.

Time management among students has received attention in computing education research. Measuring time usage or time management, by itself, can be challenging as different approaches for measuring time use (e.g. self-reported time, time logged from a learning environment) may not correlate strongly [9]. It has been suggested that surveys used for studying students' learning behaviors such as the MSLQ [132] may measure what students think they do instead of what they actually do [74], even though some of the metrics correlate with course success [107]. Despite the difficulty in measuring time usage, it has been shown that the amount of time that students spend on exercises tends to have an effect on exercise scores, in general [9, 133], and the way how students space their time when working on assignments can contribute to course outcomes [74]. One of the concerns related to students' time management behavior is procrastination, i.e., students delaying the start of working on course projects despite knowing that it might lead to worse outcomes. When

studying student data over multiple years, edwards2009comparing observed that students starting their project work late in general tend to earn poorer grades than those starting early.

Researchers have shown various ways that can influence when students start their work. These include procrastination interventions such as e-mail alerts [110], dashboards or visualizations that illustrate how students are managing their time and may implicitly set objectives on time management to students [108, 134], and more generic course design principles, such as offering practice tasks before larger projects, which are easier to start and consequently led to course work being started earlier [135].

As shown, students' use of time has received some attention from the computing education research domain, but time management has not been studied extensively from the theory perspective [136], and, to our knowledge, there has been no work on student chronotypes in the computing education research context.

4.3 Methodology

4.3.1 Context and Data

We collected keystroke data in two contexts. For simplicity we refer to the contexts as *US* and *European*. However, as we describe below, differences between the contexts are more far-reaching than just the geographic location. Differences in the contexts are summarized in Table 4.1.

US

Keystroke data was collected during the first five weeks of a Python-based CS1 course at a mid-sized public university in the US over the course of two semesters. The custom IDE logged timestamped keystrokes from weekly programming projects. After the five weeks students transitioned to a mainstream Python IDE that did not support keystroke logging capability. 525 students participated in the study. There were three sections of the course each semester, all taught by the same instructor except for one section in the

first semester. All classes were held on Mondays, Wednesdays, and Fridays. Students were required to attend a weekly recitation where students met in smaller groups with a Teaching Assistant. During the five weeks there were five programming assignments (A1,A2,...,A5), one due each Friday at midnight. Assignments could be turned in up to one day late for a 25% penalty and each student could turn in one assignment up to two days late without penalty. The assignments were a mix of standard text-based programming projects (e.g. mortgage calculator) and turtle-based graphics projects (e.g. animate a dartboard game). The projects were manually assessed and were each worth 100 points. After three weeks a first midterm exam was administered with a second midterm exam given four weeks later. The exams were administered on a computer in a testing center and included multiple choice and true/false questions. Some simple fill in the blank questions were included, such as, "what does the following code output to the screen?"

European

Keystroke data was collected for the duration of a 7-week Java-based CS1 course at a large public research-first university in Finland. The course follows a pedagogy where students work on tens of programming exercises each week, including exercises with multiple graded steps realized through intermediate goals (or "subgoals" [137]). Exercises and course materials were released weekly, and students had approximately one week to complete the exercises for a particular week. The exercise handouts were embedded in the course materials so that whenever a new topic was learned, students immediately saw programming exercises that they could work on to internalize the topic.

Exercises were worked on in a custom desktop IDE that kept track of the exercise that the student was working on and provided support for testing and submitting the exercises. In addition, the IDE collected timestamped keystrokes, used for both detecting plagiarism [23, 55] and for course improvement and research purposes. The exercises were automatically assessed and submitting work for grading after the deadline was not possible.

The course under study had one weekly lecture, held on Tuesdays, and walk-in labs, which were open daily. In the walk-in labs, students were guided by the course teacher

Table 4.1: Summary of contexts. Daylight is measured on January 1.

	US	European
Daylight (hrs)	9	6
Language (prog.)	Python	Java
Language (inst.)	English	Finnish
Participants	519	318
# keystrokes	11,185,716	20,081,066
Due dates	flexible	fixed
Projects	10	147
Exam	Midterm	Midterm & Final

and course assistants. There were two exams, one in the middle of the course and one at the end of the course. The exams were computer-based and contained programming tasks similar to the ones that students had worked on during the course. Grading-wise, 70% of the course score comes from completed exercises, and 30% from the exams.

For the analysis conducted in this article, we focus on students who attended at least two weeks of the course. This is due to the university allowing sampling courses (for no fee), and withdrawing with no repercussions.

4.3.2 Research Questions

Our research questions for this study are as follows.

RQ1 What chronotypes would be discovered from clustering keystroke data collected from two contexts?

RQ2 How do the typical working times of students relate to academic outcomes?

RQ3 How do contextual factors affect academic outcomes?

4.3.3 Metrics

We analyze distributions of keystroke timestamps. In order to identify student chronotypes from the data we first assign a 4-dimensional feature vector to each student, where the vector is the percentage of the student's keystrokes that occurred in different ranges of hours: [3-9, 9-15, 15-21, 21-3]. So a student with a feature vector of, say, [0.08, 0.1, 0.19, 0.63] types 63% of their keystrokes between 21:00 and 3:00, so they would likely be classified

as having an evening or night chronotype. After computing feature vectors for each student we used k-means clustering to identify student chronotypes. Each of the two contexts (US and European) were clustered separately. We used the elbow method to decide k. Two reasonable candidate values of k for the US context were 4 and 8, while for European, 3 and 5 were the best. We chose to use $k = 4$ in order to have a consistent value for both the US and European contexts and because it allowed us to compare our results directly with those of Putilov2019, who proposed four chronotypes. After identifying clusters of students, we study distributions of course outcomes per identified chronotype, including assignment scores, exam scores, hours before deadline, and number of keystrokes.

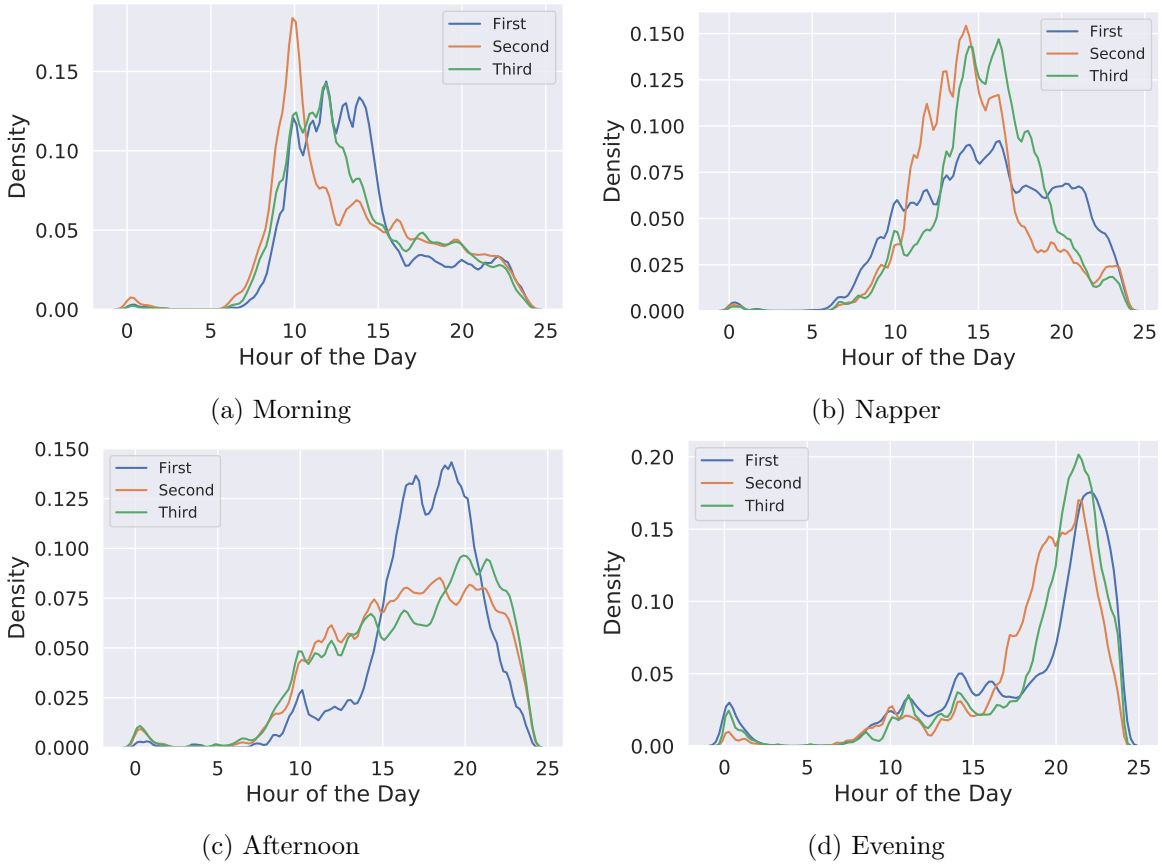
The hour ranges for the feature vector are chosen arbitrarily. As such, we must take care to ensure that this particular selection doesn't introduce bias into the analysis. Ideally we would bin keystrokes into 24 one-hour bins, but clustering on 24-dimensional vectors suffers from the curse of dimensionality. Instead, we bin student keystrokes twice more using shifted hour ranges of [5-11, 11-17, 17-23, 23-5] and [7-13, 13-19, 19-1, 1-7]. We then cluster students based on these feature vectors and compare the resulting clusters across the three binning strategies. Similarity of distributions indicates robustness to binning strategy whereas distributions that are not similar indicate a bias caused by the selected bins.

All programming project keystrokes during the study period in the US context were used in our analyses. In the European context, when analyzing the hours before deadline, we excluded keystrokes from weeks 4 and 7. During week 4 there were technical issues with the assessment system and the weekly deadlines were prolonged, while in week 7 students had two weeks for completing the assignments instead of the normal one week.

In our analysis we performed 16 statistical significance tests, 8 of them Kruskal-Wallis H, 4 Mann-Whitney U, and 4 chi-squared. We report p-values of our statistical tests as one component among others that together contribute to understanding of our results [94]. We do not make threshold-based claims of statistical significance [138].

4.3.4 Ethics

Data from the United States was collected and analyzed under Utah State University



(a) Morning

(b) Napper

(c) Afternoon

(d) Evening

Fig. 4.1: Comparison of keystroke distributions for each proposed chronotype across feature vector definitions for the US dataset. Vector binning strategies are: First = [3-9, 9-15, 15-21, 21-3]; Second = [5-11, 11-17, 17-23, 23-5]; Third = [7-13, 13-19, 19-1, 1-7].

IRB 9580. Data from the European context was collected and analyzed with student consent according to the ethical protocols outlined by The Finnish National Board on Research Integrity TENK[1].

4.4 Results

4.4.1 Comparison of binning strategies

Fig. 4.1 shows keystroke distributions for each proposed chronotype across feature vector definitions for the US context. In the clustering labeled "First" we described each

[1]https://tenk.fi/en/ethical-review/ethical-review-human-sciences

Table 4.2: Statistics of the clusters for the two contexts, US and European. The four *Centroid* columns indicate the 4D centroid discovered in k-means clustering. *Students* is the number of students in each cluster. *Keystrokes* is the median number of keystrokes per student. *Homework average score* is on project A5 (max of 100 points) for the US context and on all homework programming assignments (max of 215 points) for the European context. *Exam average score* is on the first midterm (max of 100 points) for the US context and is a combined score on the two exams (max of 30 points combined) for the European context. *Hours before deadline* is calculated by finding the median keystroke timestamp for each week and averaged over all weeks. Standard deviations are included with the homework/exam scores and hours before the deadline averages.

		Centroid						Avg score		Hours
Context	Cluster	3-9	9-15	15-21	21-3	Students	Keystrokes	Homework	Exam	before deadline
US	morning	0.03	0.70	0.19	0.08	24% (123)	21912	84 ± 28	79 ± 12	35 ± 32
European	morning	0.02	0.67	0.27	0.04	11% (36)	58992	162 ± 56	22 ± 11	85 ± 36
US	napper	0.06	0.39	0.43	0.12	40% (208)	25078	83 ± 29	77 ± 12	33 ± 36
European	napper	0.01	0.42	0.44	0.12	31% (98)	70949	172 ± 47	24 ± 8	59 ± 34
US	afternoon	0.01	0.12	0.76	0.11	21% (111)	18008	73 ± 35	74 ± 14	21 ± 30
European	afternoon	0.01	0.19	0.60	0.20	29% (91)	60427	165 ± 48	25 ± 9	46 ± 38
US	evening	0.01	0.16	0.29	0.53	15% (77)	17970	70 ± 36	71 ± 15	14 ± 29
European	evening	0.03	0.15	0.35	0.46	29% (93)	53595	159 ± 57	22 ± 11	46 ± 37

student with the four-dimensional vector used in the remainder of our analysis, with percentage of keystrokes occurring in each of four 6-hour time intervals starting at 3:00am. The "Second" clustering used time intervals starting at 5:00am and the "Third" started at 7:00am.

4.4.2 Chronotypes learned from clustering

Our first research question is: *What chronotypes would be discovered from clustering keystroke data collected from two contexts?* Related questions include whether the chronotypes between the two contexts match each other and whether they match those found in the literature.

Fig. 4.2 shows keystroke distributions for each of the four clusters for each of the two contexts. It is notable how similar the four clusters are between the two contexts. Considering the differences in context, including country, latitude (over 18 deg difference), average daylight (roughly 3 hours), instructor, course structure, homework composition, course pacing, and class time, the distributions are remarkably similar. Indeed, hours remaining before due date and number of keystrokes (Fig. 4.3) show similar behaviors across clusters even given the context differences.

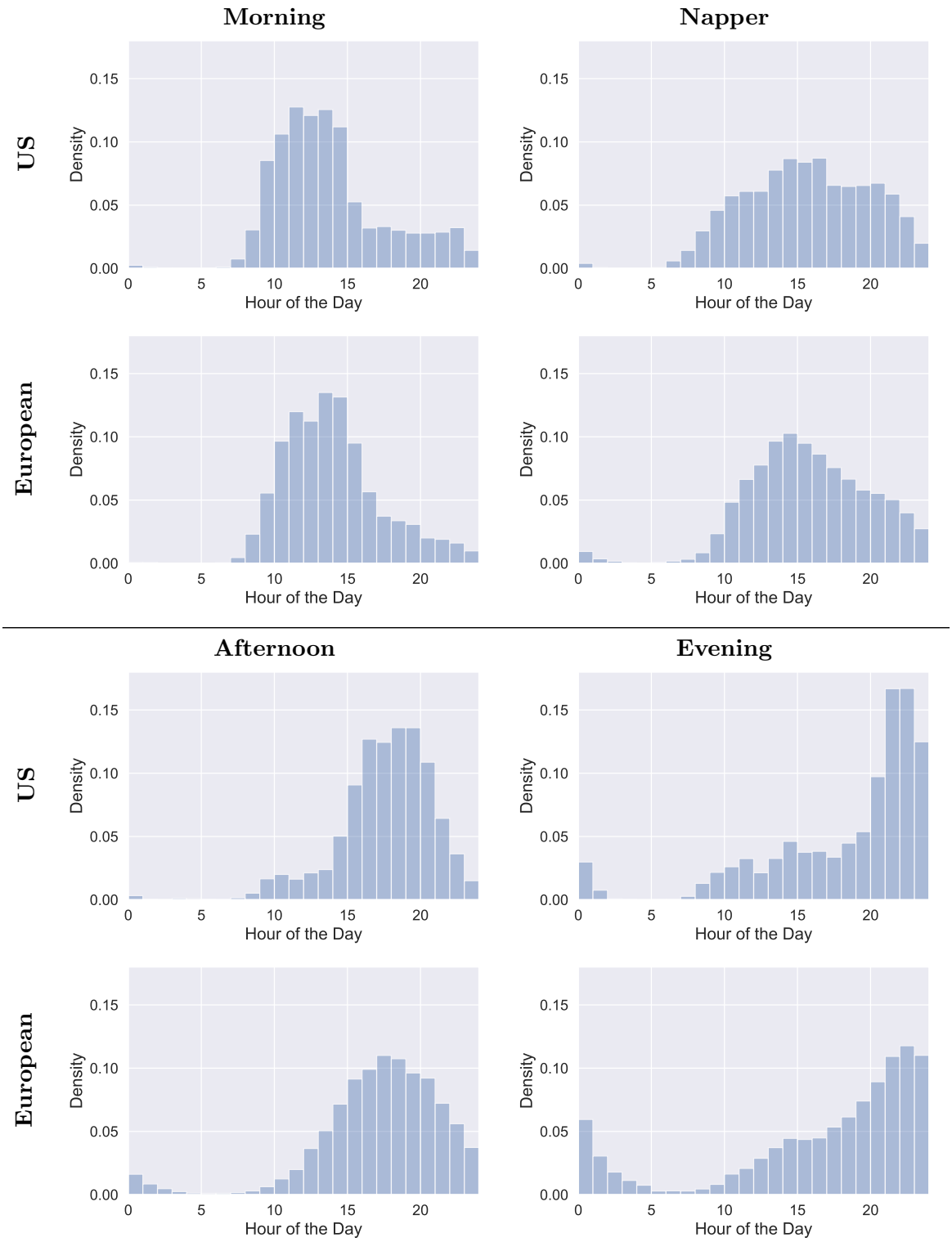


Fig. 4.2: Distribution of keystrokes for the four clusters for each context labeled with proposed chronotypes.

Due to distinctive differences between the discovered clusters and their high similarity to the Putilov's et al. [120] chronotypes, we assigned the same terms to the clusters (Fig. 4.2). For the morning cluster, peak activity is from 10am to 3pm with activity trailing off into the evening. The napper cluster has the widest range of active times. The afternoon cluster has peak activity from 1pm to 5pm, but activity is also reasonably high from 10am to 1pm. The evening cluster starts around 9am with peak activity from 8pm to 11pm.

4.4.3 Chronotype and relation to course outcomes

In this section we explore the following research question: *How do the typical working times of students relate to academic outcomes?* In the following discussion, refer to Table 4.3 for results of Kruskal-Wallis H significance testing between clusters. We first look at correlations of chronotype with project and exam scores. In Fig. 4.3 (*Projects*) we see a pronounced difference in median project score between the four chronotypes in the US context but much less so in the European context, and indeed, there are strongly significant differences in the US context and no statistically detectable differences in the European context (Table 4.3). We see similar phenomena relating to exam scores in Fig. 4.3 (*Exams*): for the US context, strongly significant differences exist whereas the European context had almost no difference, although the European context may have been influenced by the ceiling effect in this case. For both project and exam scores in the US context, morning and napper students performed better than afternoon and evening students.

We also considered the correlation between chronotype and how long before the due date students work on their assignments (*Hours remaining* in Fig. 4.3). Both contexts showed strongly significant differences, with students who work on their assignments in the morning generally working earlier relative to the due date than students who work in the evening and night time. Similarly, Fig. 4.3 (*# keystrokes*) shows the number of keystrokes executed by students. Students in the evening chronotype typed fewer characters, on average, than students in the other chronotypes. We discuss implications of these results in Section 4.5.

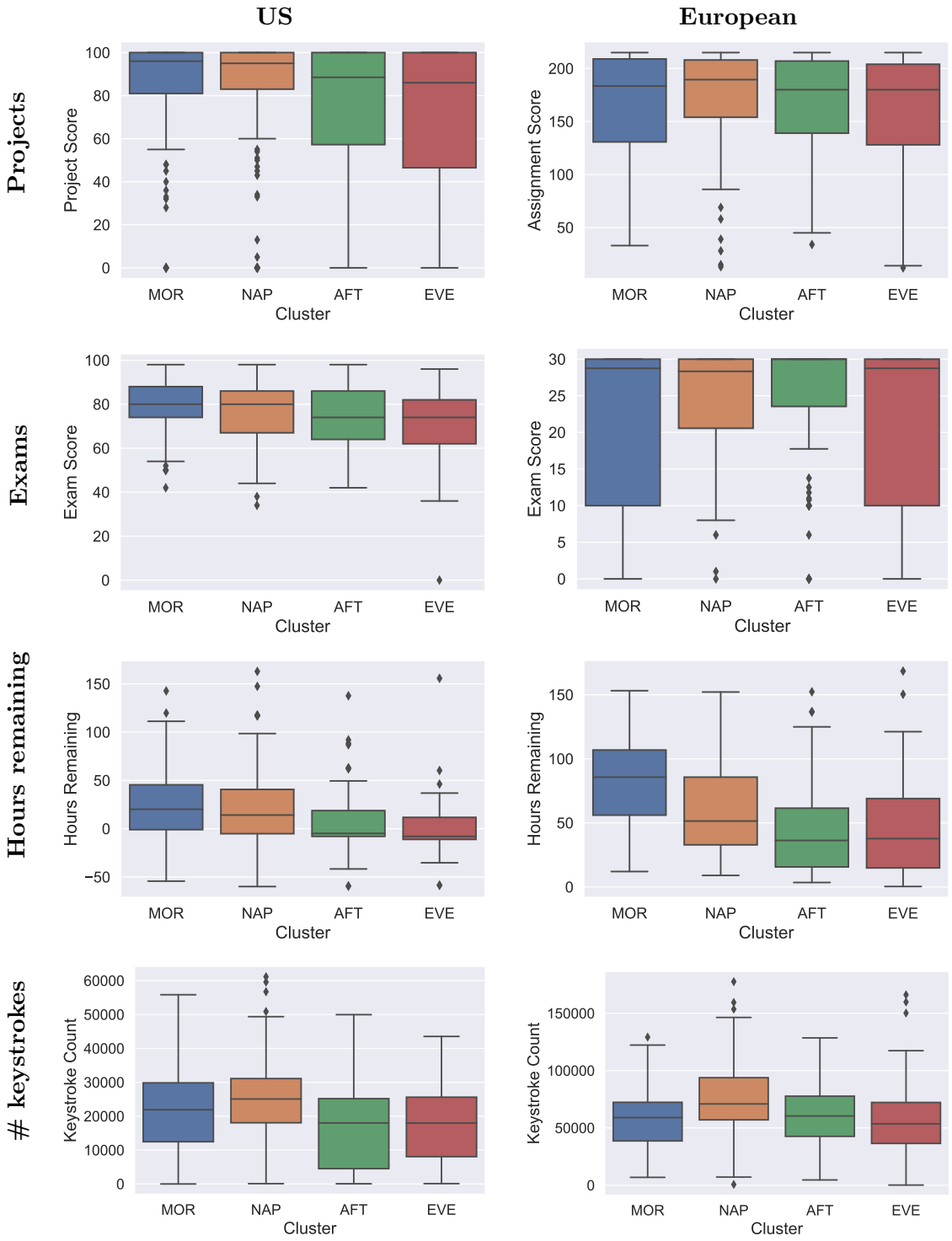


Fig. 4.3: Outcomes by cluster: project score, exam score, hours remaining before due date, and number of keystrokes. Whiskers extend 1.5·IQR past the low and high quartiles.

4.4.4 Differences in context

In this section we report results of the research question: *How do contextual factors affect academic outcomes?* We first note from Table 4.2 that, while the centroids of the clusters are remarkably similar between the US and European contexts, the distribution of students among the clusters is different. The majority of US students have morning and napper chronotypes, whereas most European students are afternoon and evening. A number of factors could be causing this difference, including culture, demographic, and hours of sunlight per day (Table 4.1). Further investigation is needed to understand the cause of the differences in distribution.

One of the most notable differences between the US and the European contexts is the time before the due date of the keystrokes. See Fig. 4.3 (*Hours remaining*). For two of the US clusters, the median keystroke is actually *later* than the due date (note from Table 4.2 that the mean, however, is before the due date), while the median keystrokes for all four clusters in the European context are well before the due date. From Table 4.2 we see that, in every chronotype, students in the European context work on their projects at least 25 hours earlier, on average, than students in the US context, and European morning students work fully 50 hours earlier on average. We propose two contextual differences contributing to this effect. The first is the late-work policy. In the US context students could turn in late work for reduced points. In the European context, late work was not accepted. The second factor is project composition. It has been shown that when projects are broken into multiple smaller pieces, students tend to start earlier [135]. Indeed, our results support this conclusion: in the US context the projects are broken into two sub-projects, while the European context has tens of sub-projects due each week. Considering the factors of late-work policy and multiple, smaller projects, it isn't surprising that a difference in time before the due date exists between the contexts. What is perhaps surprising, however, is the magnitude of the difference.

4.5 Discussion

Table 4.3: Results of running a Kruskal-Wallis H significance test across clusters for different outcomes.

	US		European	
	H	p	H	p
Assignment scores	13.178	0.004	2.386	0.496
Exam	18.894	< 0.001	4.657	0.199
Hours remaining	58.162	< 0.001	35.667	< 0.001
# keystrokes	40.947	< 0.001	21.054	< 0.001

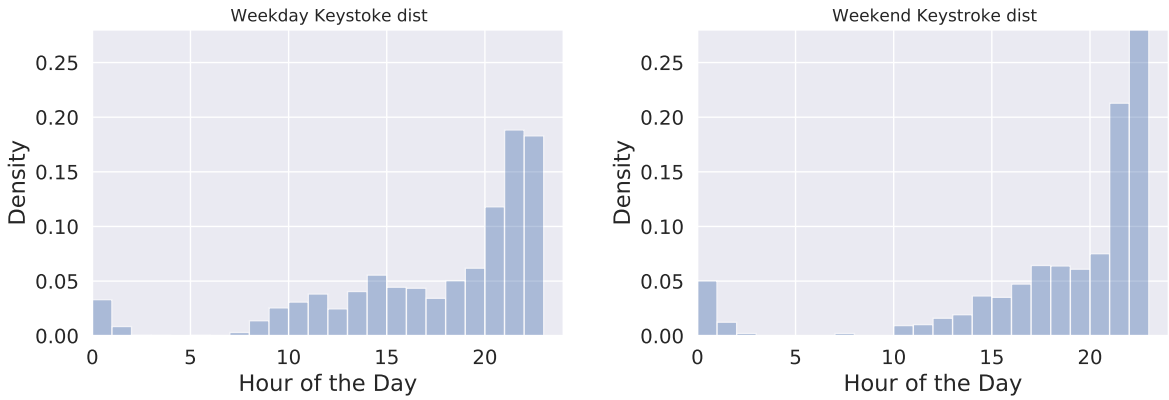
4.5.1 Robustness of clustering

In order to avoid the fragility of clustering in higher dimensions, we clustered four-dimensional feature vectors into chronotypes. To mitigate the risk of biasing the clusters by quantization strategy we clustered two additional times with different bins. Fig. 4.1, which shows distributions of keystrokes for each proposed chronotype across feature vector definitions, indicates that clusters of students are reasonably robust to binning strategy.

4.5.2 Robustness to external factors

Many factors beyond chronotype have the potential to influence how students use their time, including other courses, jobs, family obligations, social engagements, and recreation. The extent to which these external factors influence when students work on their programming assignments and whether discovered clusters indeed represent students' chronotypes are two important questions. For example, we claim that students in our *evening* group are of the evening chronotype, but one could claim that they are actually students with day jobs and are thus constrained to complete their programming assignments in the evenings.

Looking separately at working days and weekends (Fig. 4.4) – two time periods that generally have little in common with respect to jobs, class times, and other responsibilities – we see that external factors have little effect on when students work on their assignments: patterns of activity times remain reasonably constant. Using our previous example, evening students tended to work on their assignments in the evening whether it was during the week or on the weekend, suggesting that even though they may have the option to work on homework in the morning during weekends, they still choose to do so in the evening, an



(a) Weekday (b) Weekend

Fig. 4.4: Keystroke distribution of the evening chronotype in the US split into weekday and weekend keystrokes.

indication that times students work on their assignments are governed primarily by diurnal preference.

To test the similarity of the weekday and weekend distributions in Fig. 4.4 we ran Mann-Whitney U tests on the distributions for all chronotypes (only the evening chronotype is shown in the figure) shifted by 7 hours to make the distributions unimodal. Because the cyclic nature of the data could compromise the integrity of a rank-based test, we also ran chi-squared tests using the 24 one-hour bins shown in the figures. Both tests for each of the four chronotypes yielded $p < 0.0001$. However, all effect sizes (Cohen's d for Mann-Whitney U and Cramer's V for chi-squared) were 0.2 or less, so, while behavior is different between weekday and weekend, the difference is small.

4.5.3 General Insights

There is a common stereotype of computer programmers as being night owls. A striking result of our data is that, at least among CS1 students, this stereotype does not hold up. The majority of keystrokes in our study were done between 9am and 5pm (54%) and most of the remaining keystrokes were executed between 5pm and 9pm (27%). Only 1% of keystrokes occurred between midnight and 5:00am. Our results support prior work that has found that the majority of programmers follow typical working hours [111]. Indeed, even among

the few students in the evening chronotype, most keystrokes were logged before midnight.

Another important result is that, while context appeared to have a strong effect on correlations between chronotype and outcomes, the chronotypes themselves found through unsupervised learning were very similar between the two contexts. The fact that similar chronotypes were found, both in terms of cluster centroids (Table 4.2) and keystroke distributions (Fig. 4.2) between two very different contexts is a strong support for chronotype theory and its characterizations [120].

4.5.4 Effect on course outcomes

A result exhibited in both the US and European contexts is that morning and napper students started working on weekly assignments much earlier than afternoon and evening students and nappers typed more characters in working on their assignments. Our results also provide some support for correlations between chronotype and academic achievement [104]: in the US context the evening chronotype was associated with lower assignment and exam scores when compared to the morning chronotype. However, the European context didn't show a difference. It is possible that the ceiling effect caused a lack of difference in project and exam scores between chronotypes in the European context. It is also possible that the design of programming assignments in the European context, as discussed in Section 4.4.4, could have affected academic outcomes across chronotypes, suggesting investigation into assignments with smaller pieces and fixed due dates.

4.5.5 Synchrony with chronotype

Research has found that scheduling learning activities during students' preferred working times increases academic achievement [123,124]. Our data shows afternoon and evening students performing, in general, at a lower level. From the synchrony effect theory we consider the possibility that the university class schedule forced these students to work at suboptimal times, contributing to their sub-par behavior. To see if the imposed schedule affected their natural working time we compare each chronotype's keystroke distributions split between weekdays and weekends (the weekday/weekend distributions for the evening

chronotype in the US context are shown in Fig. 4.4), as students would be expected to be more free to work according to their preferred schedule on weekends. As can be seen in the figure, students behaved very similarly on the weekends as they did during the week. Because of the small effect size (0.2, as reported above in Section 4.5.2), which suggests that students worked similar hours whether during the week or on weekends, it appears that synchrony had only minimal effect on afternoon and evening student underachievement.

4.5.6 Limitations

We did not collect demographic or background information about study participants, which means that students who work at jobs or have other classes may have affected our chronotype distributions. However, the effect of these external factors appears to be minimal (see Section 4.5.2). Furthermore, Putilov2019 indicated that there is a group of people that doesn't belong to any of the four clusters and these classification-defying subjects may have diluted clarity of our clusters.

4.6 Conclusion

In this article, we analyzed evidence for the existence of chronotypes using keystroke data collected from introductory programming students. To summarize, our research questions and their answers are as follows.

(RQ1) "What chronotypes would be discovered from clustering keystroke data collected from two contexts?" We identified four chronotypes similar to those discussed in the literature [120]: morning, napper, afternoon, and evening. These clusters were identified in both studied contexts, despite the differences in how the courses were organized and, e.g., the amount of available daylight. It also seems that these distributions are not significantly influenced by flexible due dates or deadlines.

(RQ2) "How do the typical working times of students relate to academic outcomes?" We observed noticeable differences in the exam scores and project scores within the US context, where those active in the morning performed the best in the exam. No significant differences in exam scores or project scores were observed in the European context, although

this could be partially influenced by a ceiling effect. In both contexts, the morning and napper chronotypes tended to start working on their projects earlier than the afternoon and evening chronotypes, and the napper chronotypes tended to type, in general, more than the other chronotypes.

(RQ3) "How do contextual factors affect academic outcomes?" We observed differences between the contexts in when the students started to work on their projects and noticed that, in general, students in the European context tended to start their work earlier. We hypothesized that this could be due to two factors; (1) using small exercises when starting with a new topic, and (2) having a strict no-late submissions policy. While we acknowledge that there are likely many other explanations for these observations, we posit that the way how courses are organized can lead to a situation where students are more likely to follow their diurnal preferences, i.e., work during times that are productive for them.

In this article we have inferred behavior from observed keystrokes and while our conclusions are in line with prior theoretical and empirical research, we cannot for certain say whether our observations stem from students' circadian rhythms and diurnal preferences, or whether there are other factors at play. We do not know, for example, how many of the students have part-time jobs and how their working hours are distributed over the week. Regardless, we observe that the majority of the students tend to work during daylight hours, contrary to some of the stereotypes posited about CS students. And, more importantly, the chronotypes we discover and their correlations with academic outcomes align with prior studies, suggesting that we, as computing education researchers and practitioners, should take note and consider diurnal preference issues in course development and future research.

As a part of our future work, we are looking into the translation and use of a chronotype questionnaire (e.g. Caen Chronotype Questionnaire [139]) and studying whether the chronotypes identified from keystroke data match those identified using a questionnaire. As previous research suggests that working on optimal hours of day contribute towards performance in complex tasks [130], we are also looking into the interplay of task difficulty,

chronotype, observed behavior, and performance, within the domain of learning programming.

4.7 Data Availability

Our human-centered study protocols prohibit us from making our keystroke data publicly available. Our code, however, is available at http://doi.org/10.5281/zenodo.4498457.

CHAPTER 5

Conclusion

In this thesis we have discussed how keystroke data can be useful in understanding programming process, identifying patterns and looking into behaviors of CS1 students. These insights can be valuable in predicting struggling students and working on targeted interventions which can boost students' confidence and knowledge to decrease the attrition rates in computer science students.

Chapter 2 presents a work and a tool for visualizing the programming process. We have shown that instructors can use this tool to augment their assessment, identify approaches that students are using to work on their solution, identify cases of plagiarism and also indicate whether a student is struggling on the assignment. It can be also used for pair-programming and self-reflection among students. Further improvement and work on this tool can make it fully autonomous to report students and their processes to instructors.

Chapter 3 presents a work to understand and analyze pausing patterns of CS1 students and its implications on academic outcomes. We observe that the pauses a student takes has a correlation with their performance (exam score). Similarly we also find populations among student based on their pausing behaviors, shorter and longer pause group, and see some correlation between typical pause lengths and exam scores. We also find evidence that students are not pausing randomly. Students tend to pause after finishing their thoughts and pause after a natural stopping point. We suggest the students pausing after pressing delete key are possibly less engaged. Students pausing after special characters and failed compiles suggest at risk students and further pedagogical and material innovations can be used to improve student fluency and minimize the failed runs.

Chapter 4 presents a work on student behaviors and suggests the evidence of chronotypes on introductory programming students. The clustering analysis discovered four different groups, morning, napper, afternoon, and evening, that are similar to the groups in

the literature. We also observed that there are noticeable differences in the exam scores and project scores between these groups and we found that the students working in the morning are usually performing better in the exam than other groups. Similarly, we also find differences in student behavior based on the context and different type of assignments. We found that the students tend to start their work earlier when using small exercises when compared to large assignments. This information can be useful in designing courses to reduce procrastination and increase academic outcomes.

REFERENCES

[1] C. Watson and F. W. Li, "Failure rates in introductory programming revisited," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, 2014, pp. 39–44.

[2] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming," *AcM SIGcSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007.

[3] A. Vihavainen, J. Airaksinen, and C. Watson, "A systematic review of approaches for teaching introductory programming and their influence on success," in *Proceedings of the tenth annual conference on International computing education research*, 2014, pp. 19–26.

[4] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the second international workshop on Computing education research*, 2006, pp. 73–84.

[5] C. Watson, F. W. Li, and J. L. Godwin, "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, 2013, pp. 319–323.

[6] A. S. Carter, C. D. Hundhausen, and O. Adesope, "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*, 2015, pp. 141–150.

[7] R. Smith and S. Rixner, "The error landscape: Characterizing the mistakes of novice programmers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 538–544.

[8] E. Kurvinen, N. Hellgren, E. Kaila, M.-J. Laakso, and T. Salakoski, "Programming misconceptions in an introductory level programming course exam," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 2016, pp. 308–313.

[9] J. Leinonen, L. Leppänen, P. Ihantola, and A. Hellas, "Comparison of time metrics in programming," in *Proceedings of the 2017 acm conference on international computing education research*, 2017, pp. 200–208.

[10] M. C. Jadud and B. Dorn, "Aggregate compilation behavior: Findings and implications from 27,698 users," in *Proceedings of the eleventh annual international conference on international computing education research*, 2015, pp. 131–139.

[11] K. Longi, J. Leinonen, H. Nygren, J. Salmi, A. Klami, and A. Vihavainen, "Identification of programmers from typing patterns," in *Proceedings of the 15th Koli Calling conference on computing education research*, 2015, pp. 60–67.

[12] G. Rodriguez-Rivera, J. Turkstra, J. Buckmaster, K. LeClainche, S. Montgomery, W. Reed, R. Sullivan, and J. Lee, "Tracking large class projects in real-time using fine-grained source control," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, 2022, pp. 565–570.

[13] A. Selvaraj, E. Zhang, L. Porter, and A. G. Soosai Raj, "Live coding: A review of the literature," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 2021, pp. 164–170.

[14] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli calling international conference on computing education research*, 2010, pp. 86–93.

[15] R. S. Pettit, J. D. Homer, K. M. McMurry, N. Simone, and S. A. Mengel, "Are automated assessment tools helpful in programming courses?" in *2015 ASEE Annual Conference & Exposition*, 2015, pp. 26–230.

[16] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers *et al.*, "Educational data mining and learning analytics in programming: Literature review and case studies," *Proceedings of the 2015 ITiCSE on Working Group Reports*, pp. 41–63, 2015.

[17] J. Leinonen, "Keystroke data in programming courses," Ph.D. dissertation, University of Helsinki, 2019.

[18] A. Vihavainen, M. Luukkainen, and P. Ihantola, "Analysis of source code snapshot granularity levels," in *Proceedings of the 15th annual conference on information technology education*, 2014, pp. 21–26.

[19] B. A. Becker, "A new metric to quantify repeated compiler errors for novice programmers," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 2016, pp. 296–301.

[20] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen, "Exploring machine learning methods to automatically identify students in need of assistance," in *Proceedings of the eleventh annual international conference on international computing education research*, 2015, pp. 121–130.

[21] K. Castro-Wunsch, A. Ahadi, and A. Petersen, "Evaluating neural networks as a method for identifying students in need of assistance," in *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, 2017, pp. 111–116.

[22] J. Leinonen, K. Longi, A. Klami, and A. Vihavainen, "Automatic inference of programming performance and experience from typing patterns," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 132–137.

[23] J. Leinonen, K. Longi, A. Klami, A. Ahadi, and A. Vihavainen, "Typing patterns and authentication in practical programming exams," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 2016, pp. 160–165.

[24] J. Edwards, J. Leinonen, and A. Hellas, "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 Symposium on Computer Science Education*, 2020, pp. 413–419.

[25] A. Zavgorodniaia, R. Shrestha, J. Leinonen, A. Hellas, and J. Edwards, "Morning or evening? an examination of circadian rhythms of cs1 students," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 2021, pp. 261–272.

[26] P. Peltola, V. Kangas, N. Pirttinen, H. Nygren, and J. Leinonen, "Identification based on typing patterns between programming and free text," in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 2017, pp. 163–167.

[27] J. Edwards, J. Leinonen, C. Birthare, A. Zavgorodniaia, and A. Hellas, "Programming versus natural language: On the effect of context on typing in cs1," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, 2020, pp. 204–215.

[28] A. Vihavainen, J. Helminen, and P. Ihantola, "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*, 2014, pp. 109–116.

[29] J. Ditton, H. Swanson, and J. Edwards, "External imagery in computer programming," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 2021, pp. 1226–1231.

[30] R. L. Novais, A. Torres, T. S. Mendes, M. Mendonça, and N. Zazworka, "Software evolution visualization: A systematic mapping study," *Information and Software Technology*, vol. 55, no. 11, pp. 1860–1883, 2013.

[31] A.-L. Mattila, P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen, and H. Väätäjä, "Software visualization today: Systematic literature review," in *Proceedings of the 20th International Academic Mindtrek Conference*, 2016, pp. 262–271.

[32] L. Voinea, A. Telea, and J. J. Van Wijk, "Cvsscan: visualization of code evolution," in *Proceedings of the 2005 ACM symposium on Software visualization*, 2005, pp. 47–56.

[33] Y. Matsuzawa, K. Okada, and S. Sakai, "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*, 2013, pp. 46–51.

[34] K. Heinonen, K. Hirvikoski, M. Luukkainen, and A. Vihavainen, "Using codebrowser to seek differences between novice programmers," in *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014, pp. 229–234.

[35] S. Diehl, *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[36] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, "Experiences with marmoset: designing and using an advanced submission and testing system for programming courses," *ACM Sigcse Bulletin*, vol. 38, no. 3, pp. 13–17, 2006.

[37] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita, "Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from hackystat-uh," in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.* IEEE, 2004, pp. 136–144.

[38] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014, pp. 223–228.

[39] C. D. Hundhausen, D. M. Olivares, and A. S. Carter, "Ide-based learning analytics for computing education: a process model, critical review, and research agenda," *ACM Transactions on Computing Education (TOCE)*, vol. 17, no. 3, pp. 1–26, 2017.

[40] A. M. Kazerouni, S. H. Edwards, T. S. Hall, and C. A. Shaffer, "Deveventtracker: Tracking development events to assess incremental development and procrastination," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 104–109.

[41] M. R. A. Wittmann, M. Bower, and M. Kavakli-Thorne, "Using the score software package to analyse novice computer graphics programming," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 118–122.

[42] D. Toll and A. Wingkvist, "Visualizing programming session timelines," in *Proceedings of the 11th International Symposium on Visual Information Communication and Interaction*, 2018, pp. 106–107.

[43] D. Toll, "Measuring programming assignment effort," Ph.D. dissertation, Faculty of Technology, Linnaeus University, 2016.

[44] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831–838.

[45] J. Helminen, P. Ihantola, V. Karavirta, and L. Malmi, "How do students solve parsons programming problems? an analysis of interaction traces," in *Proceedings of the ninth annual international conference on International computing education research*, 2012, pp. 119–126.

[46] C. Piech, M. Sahami, J. Huang, and L. Guibas, "Autonomously generating hints by inferring problem solving policies," in *Proceedings of the second (2015) acm conference on learning@ scale*, 2015, pp. 195–204.

[47] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein, "Modeling how students learn to program," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012, pp. 153–160.

[48] C. Murphy, G. Kaiser, K. Loveland, and S. Hasan, "Retina: helping students and instructors based on observed programming activities," in *Proceedings of the 40th ACM technical symposium on Computer Science Education*, 2009, pp. 178–182.

[49] C. Norris, F. Barry, J. B. Fenwick Jr, K. Reid, and J. Rountree, "Clockit: collecting quantitative data on how beginning software developers really work," in *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, 2008, pp. 37–41.

[50] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "Fastdash: a visual dashboard for fostering awareness in software teams," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 1313–1322.

[51] P. Ihantola and A. Petersen, "Code complexity in introductory programming courses," in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[52] E. Balzuweit and J. Spacco, "Snapviz: visualizing programming assignment snapshots," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, 2013, pp. 350–350.

[53] A. Ahadi, R. Lister, and L. Mathieson, "Aral: An online tool for source code snapshot metadata analysis," in *Proceedings of the Twenty-First Australasian Computing Education Conference*, 2019, pp. 118–125.

[54] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.

[55] A. Hellas, J. Leinonen, and P. Ihantola, "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*, 2017, pp. 238–243.

[56] M. C. Jadud, "A first look at novice compilation behaviour using bluej," *Computer Science Education*, vol. 15, no. 1, pp. 25–40, 2005.

[57] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 522–527. [Online]. Available: https://doi.org/10.1145/2676723.2677258

[58] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 75–80. [Online]. Available: http://doi.acm.org/10.1145/2325296.2325318

[59] A. Vihavainen, J. Helminen, and P. Ihantola, "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*, ser. Koli Calling '14. New York, NY, USA: ACM, 2014, pp. 109–116. [Online]. Available: http://doi.acm.org/10.1145/2674683.2674692

[60] A. Kołakowska, "Towards detecting programmers' stress on the basis of keystroke dynamics," in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2016, pp. 1621–1626.

[61] R. C. Thomas, A. Karahasanovic, and G. E. Kennedy, "An investigation into keystroke latency metrics as an indicator of programming performance," in *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, 2005, pp. 127–134.

[62] J.-N. Foulin, "Pauses et débits : les indicateurs temporels de la production écrite," *L'année psychologique*, vol. 95, no. 3, pp. 483–504, 1995.

[63] J. Cenoz, "Pauses and hesitation phenomena in second language production," *ITL - International Journal of Applied Linguistics*, vol. 127-128, pp. 53–69, Jan. 2000.

[64] A. Révész, M. Michel, and M. Lee, "EXPLORING SECOND LANGUAGE WRITERS' PAUSING AND REVISION BEHAVIORS," *Studies in Second Language Acquisition*, vol. 41, no. 3, pp. 605–631, Jul. 2019.

[65] M. Kumpulainen, "On the operationalisation of 'pauses' in translation process research," *Translation & Interpreting*, vol. 7, no. 1, pp. 47–58, 2015.

[66] S. O'Brien, "Pauses as indicators of cognitive effort in post-editing machine translation output," *Across Languages and Cultures*, vol. 7, no. 1, pp. 1–21, Jun. 2006.

[67] J. Schilperoord, *It's about time: Temporal aspects of cognitive processes in text production*. Rodopi, 1996, vol. 6.

[68] I. Lacruz and G. M. Shreve, "Pauses and cognitive effort in post-editing," in *Post-editing of Machine Translation: Processes and Applications*. Cambridge Scholars Publishing, 2014.

[69] S. P. Davies, "The role of notation and knowledge representation in the determination of programming strategy: a framework for integrating models of programming behavior," *Cognitive Science*, vol. 15, no. 4, pp. 547–572, 1991.

[70] R. S. Rist, "Schema creation in programming," *Cognitive Science*, vol. 13, no. 3, pp. 389–414, 1989.

[71] J. J. Van Merrienboer and F. G. Paas, "Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice," *Computers in Human Behavior*, vol. 6, no. 3, pp. 273–289, 1990.

[72] A. Altadmri, M. Kolling, and N. C. Brown, "The cost of syntax and how to avoid it: Text versus frame-based editing," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2016, pp. 748–753.

[73] A. Petersen, J. Spacco, and A. Vihavainen, "An exploration of error quotient in multiple contexts," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 2015, pp. 77–86.

[74] L. Leppänen, J. Leinonen, and A. Hellas, "Pauses and spacing in learning to program," in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. ACM, 2016, pp. 41–50.

[75] M. Leinikka, A. Vihavainen, J. Lukander, and S. Pakarinen, "Cognitive flexibility and programming performance," in *Psychology of programming interest group workshop*, 2014, pp. 1–11.

[76] A. Hellas, P. Ihantola, A. Petersen, V. V. Ajanovski, M. Gutica, T. Hynninen, A. Knutas, J. Leinonen, C. Messom, and S. N. Liao, "Predicting academic performance: a systematic literature review," in *Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education*, 2018, pp. 175–199.

[77] L. Porter, D. Zingaro, and R. Lister, "Predicting student success using fine grain clicker data," in *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ser. ICER '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 51–58. [Online]. Available: https://doi.org/10.1145/2632320.2632354

[78] S. N. Liao, D. Zingaro, K. Thai, C. Alvarado, W. G. Griswold, and L. Porter, "A robust machine learning technique to predict low-performing students," *ACM Trans. Comput. Educ.*, vol. 19, no. 3, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3277569

[79] M. Tukiainen and E. Mönkkönen, "Programming aptitude testing as a prediction of learning to program." in *PPIG*, 2002, p. 4.

[80] L. H. Werth, "Predicting student performance in a beginning computer science class," *ACM SIGCSE Bulletin*, vol. 18, no. 1, pp. 138–143, 1986.

[81] N. Rountree, J. Rountree, A. Robins, and R. Hannah, "Interacting factors that predict success and failure in a cs1 course," *ACM SIGCSE Bulletin*, vol. 36, no. 4, pp. 101–104, 2004.

[82] J. Bennedsen and M. E. Caspersen, "Abstraction ability as an indicator of success for learning object-oriented programming?" *ACM Sigcse Bulletin*, vol. 38, no. 2, pp. 39–43, 2006.

[83] S. Bergin and R. Reilly, "Programming: factors that influence success," in *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 2005, pp. 411–415.

[84] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. A. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll, "Educational data mining and learning analytics in programming: Literature review and case studies," in *Proc. of the 2015 ITiCSE on Working Group Reports*, ser. ITICSE-WGR '15. ACM, 2015, pp. 41–63.

[85] C. Epp, M. Lippold, and R. L. Mandryk, "Identifying emotional states using keystroke dynamics," in *Proceedings of the sigchi conference on human factors in computing systems*, 2011, pp. 715–724.

[86] J. Edwards, J. Ditton, D. Trninic, H. Swanson, S. Sullivan, and C. Mano, "Syntax exercises in CS1," in *Proceedings of the 16th Annual Conference on International Computing Education Research*, ser. ICER '20, 2020.

[87] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel, "Scaffolding students' learning using test my code," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, 2013, pp. 117–122.

[88] L. V. Waes and P. J. Schellens, "Writing profiles: the effect of the writing mode on pausing and revision patterns of experienced writers," *Journal of Pragmatics*, vol. 35, no. 6, pp. 829–853, Jun. 2003.

[89] R. A. Alves, S. L. Castro, L. de Sousa, and S. Strömqvist, "Chapter 4: Influence of typing skill on pause–execution cycles in written composition," in *Writing and Cognition*. BRILL, Jan. 2007, pp. 55–65.

[90] T. Olive, R. A. Alves, and S. L. Castro, "Cognitive processes in writing during pause and execution periods," *European Journal of Cognitive Psychology*, vol. 21, no. 5, pp. 758–785, Aug. 2009.

[91] R. Revlin, *Cognition : theory and practice*. New York, NY: Worth Publishers, 2013.

[92] J. P. Borst, N. A. Taatgen, and H. van Rijn, "What makes interruptions disruptive?: A process-model account of the effects of the problem state bottleneck on task interruption and resumption," in *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. ACM, 2015, pp. 2971–2980.

[93] J. Leinonen, F. E. V. Castro, and A. Hellas, "Fine-grained versus coarse-grained data for estimating time-on-task in learning programming," in *Proceedings of The 14th International Conference on Educational Data Mining (EDM 2021)*. The International Educational Data Mining Society, 2021.

[94] R. L. Wasserstein and N. A. Lazar, "The asa statement on p-values: context, process, and purpose," 2016.

[95] W. Haynes, *Bonferroni Correction*. New York, NY: Springer New York, 2013, pp. 154–154. [Online]. Available: https://doi.org/10.1007/978-1-4419-9863-7_1213

[96] R. L. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.

[97] A. J. Gould, "What makes an interruption disruptive? understanding the effects of interruption relevance and timing on performance," Ph.D. dissertation, UCL (University College London), 2014.

[98] S. T. Iqbal and B. P. Bailey, "Leveraging characteristics of task structure to predict the cost of interruption," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006, pp. 741–750.

[99] M. B. Edwards and S. D. Gronlund, "Task interruption and its effects on memory," *Memory*, vol. 6, no. 6, pp. 665–687, 1998.

[100] T. Gillie and D. Broadbent, "What makes interruptions disruptive? a study of length, similarity, and complexity," *Psychological research*, vol. 50, no. 4, pp. 243–250, 1989.

[101] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of learning in novice programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 37–55, 1986.

[102] R. S. Rist, "Program structure and design," *Cognitive science*, vol. 19, no. 4, pp. 507–562, 1995.

[103] L. E. Winslow, "Programming pedagogy—a psychological overview," *ACM Sigcse Bulletin*, vol. 28, no. 3, pp. 17–22, 1996.

[104] F. Preckel, A. A. Lipnevich, S. Schneider, and R. D. Roberts, "Chronotype, cognitive abilities, and academic achievement: A meta-analytic investigation," *Learning and Individual Differences*, vol. 21, no. 5, pp. 483–492, 2011.

[105] D. Fucci, G. Scanniello, S. Romano, and N. Juristo, "Need for sleep: The impact of a night of sleep deprivation on novice developers' performance," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 1–19, Jan. 2020. [Online]. Available: https://doi.org/10.1109/tse.2018.2834900

[106] T. Roenneberg, A. Wirz-Justice, and M. Merrow, "Life between clocks: Daily temporal patterns of human chronotypes," *Journal of Biological Rhythms*, vol. 18, no. 1, pp. 80–90, Feb. 2003. [Online]. Available: https://doi.org/10.1177/0748730402239679

[107] S. Bergin and R. Reilly, "The influence of motivation and comfort-level on learning to program," 2005.

[108] K. Ilves, J. Leinonen, and A. Hellas, "Supporting self-regulated learning with visualizations in online learning environments," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 257–262.

[109] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola, "Comparing effective and ineffective behaviors of student programmers," in *Proceedings of the fifth international workshop on Computing education research workshop*, 2009, pp. 3–14.

[110] J. Martin, S. H. Edwards, and C. A. Shaffer, "The effects of procrastination interventions on programming project success," in *Proceedings of the eleventh annual International Conference on International Computing Education Research*, 2015, pp. 3–11.

[111] M. Claes, M. V. Mäntylä, M. Kuutila, and B. Adams, "Do programmers work at night or during the weekend?" in *Proceedings of the 40th International Conference on Software Engineering*. ACM, May 2018. [Online]. Available: https://doi.org/10.1145/3180155.3180193

[112] S. M. Reppert and D. R. Weaver, "Coordination of circadian timing in mammals," *Nature*, vol. 418, no. 6901, pp. 935–941, Aug. 2002. [Online]. Available: https://doi.org/10.1038/nature00965

[113] C. Dibner, U. Schibler, and U. Albrecht, "The mammalian circadian timing system: Organization and coordination of central and peripheral clocks," *Annual Review of Physiology*, vol. 72, no. 1, pp. 517–549, Mar. 2010. [Online]. Available: https://doi.org/10.1146/annurev-physiol-021909-135821

[114] D. Bell-Pedersen, V. M. Cassone, D. J. Earnest, S. S. Golden, P. E. Hardin, T. L. Thomas, and M. J. Zoran, "Circadian rhythms from multiple oscillators: lessons from diverse organisms," *Nature Reviews Genetics*, vol. 6, no. 7, pp. 544–556, Jul. 2005. [Online]. Available: https://doi.org/10.1038/nrg1633

[115] M. Wittmann, J. Dinich, M. Merrow, and T. Roenneberg, "Social jetlag: Misalignment of biological and social time," *Chronobiology International*, vol. 23, no. 1-2, pp. 497–509, Jan. 2006. [Online]. Available: https://doi.org/10.1080/07420520500545979

[116] A. Korczak, B. Martynhak, M. Pedrazzoli, A. Brito, and F. Louzada, "Influence of chronotype and social zeitgebers on sleep/wake patterns," *Brazilian Journal of Medical and Biological Research*, vol. 41, no. 10, pp. 914–919, Sep. 2008. [Online]. Available: https://doi.org/10.1590/s0100-879x2008005000047

[117] J. A. Vitale, E. Roveda, A. Montaruli, L. Galasso, A. Weydahl, A. Caumo, and F. Carandente, "Chronotype influences activity circadian rhythm and sleep: Differences in sleep quality between weekdays and weekend," *Chronobiology International*, vol. 32, no. 3, pp. 405–415, Dec. 2014. [Online]. Available: https://doi.org/10.3109/07420528.2014.986273

[118] M. Valladares, R. Ramírez-Tagle, M. A. Muñoz, and A. M. Obregón, "Individual differences in chronotypes associated with academic performance among chilean university students," *Chronobiology International*, vol. 35, no. 4, pp. 578–583, Dec. 2017. [Online]. Available: https://doi.org/10.1080/07420528.2017.1413385

[119] K. Porcheret, L. Wald, L. Fritschi, M. Gerkema, M. Gordijn, M. Merrrow, S. M. Rajaratnam, D. Rock, T. L. Sletten, G. Warman, K. Wulff, T. Roenneberg, and R. G. Foster, "Chronotype and environmental light exposure in a student population," *Chronobiology International*, vol. 35, no. 10, pp. 1365–1374, Jun. 2018. [Online]. Available: https://doi.org/10.1080/07420528.2018.1482556

[120] A. A. Putilov, N. Marcoen, D. Neu, N. Pattyn, and O. Mairesse, "There is more to chronotypes than evening and morning types: Results of a large-scale community survey provide evidence for high prevalence of two further types," *Personality and Individual Differences*, vol. 148, pp. 77–84, Oct. 2019. [Online]. Available: https://doi.org/10.1016/j.paid.2019.05.017

[121] S. N. Kolomeichuk, C. Randler, I. Shabalina, L. Fradkova, and M. Borisenkov, "The influence of chronotype on the academic achievement of children and adolescents – evidence from russian karelia," *Biological Rhythm Research*, vol. 47, no. 6, pp. 873–883, Jul. 2016. [Online]. Available: https://doi.org/10.1080/09291016.2016.1207352

[122] A. Montaruli, L. Castelli, L. Galasso, A. Mulè, E. Bruno, F. Esposito, A. Caumo, and E. Roveda, "Effect of chronotype on academic achievement in a sample of italian university students," *Chronobiology International*, vol. 36, no. 11, pp. 1482–1495, Aug. 2019. [Online]. Available: https://doi.org/10.1080/07420528.2019.1652831

[123] R. Dunn, "Research on instructional environments: Implications for student achievement and attitudes." *Professional School Psychology*, vol. 2, no. 1, p. 43, 1987.

[124] R. Dunn and S. A. Griggs, *Learning styles: Quiet revolution in American secondary schools.* ERIC, 1988.

[125] K. Nowack and E. Van Der Meer, "The synchrony effect revisited: chronotype, time of day and cognitive performance in a semantic analogy task," *Chronobiology international*, vol. 35, no. 12, pp. 1647–1662, 2018.

[126] C. Cajochen, S. B. S. Khalsa, J. K. Wyatt, C. A. Czeisler, and D.-J. Dijk, "EEG and ocular correlates of circadian melatonin phase and human performance decrements during sleep loss," *American Journal of Physiology-Regulatory, Integrative and Comparative Physiology*, vol. 277, no. 3, pp. R640–R649, Sep. 1999. [Online]. Available: https://doi.org/10.1152/ajpregu.1999.277.3.r640

[127] C. Schmidt, F. Collette, C. Cajochen, and P. Peigneux, "A time to think: Circadian rhythms in human cognition," *Cognitive Neuropsychology*, vol. 24, no. 7, pp. 755–789, Oct. 2007. [Online]. Available: https://doi.org/10.1080/02643290701754158

[128] P. Valdez, T. Reilly, and J. Waterhouse, "Rhythms of mental performance," *Mind, Brain, and Education*, vol. 2, no. 1, pp. 7–16, Mar. 2008. [Online]. Available: https://doi.org/10.1111/j.1751-228x.2008.00023.x

[129] L. Hasher, C. Lustig, and R. Zacks, "Inhibitory mechanisms and the control of attention," in *Variation in Working Memory.* Oxford University Press, Mar. 2008, pp. 227–249. [Online]. Available: https://doi.org/10.1093/acprof:oso/9780195168648.003.0009

[130] V. Natale and R. Lorenzetti, "Influences of morningness-eveningness and time of day on narrative comprehension," *Personality and Individual Differences*, vol. 23, no. 4, pp. 685–690, Oct. 1997. [Online]. Available: https://doi.org/10.1016/s0191-8869(97)00059-7

[131] V. Natale, A. Alzani, and P. Cicogna, "Cognitive efficiency and circadian typologies: a diurnal study," *Personality and Individual Differences*, vol. 35, no. 5, pp. 1089–1105, Oct. 2003. [Online]. Available: https://doi.org/10.1016/s0191-8869(02)00320-3

[132] P. R. Pintrich *et al.*, "A manual for the use of the motivated strategies for learning questionnaire (MSLQ)." 1991.

[133] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall, "Analyzing student work patterns using programming exercise data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 18–23. [Online]. Available: https://doi.org/10.1145/2676723.2677297

[134] T. Auvinen, L. Hakulinen, and L. Malmi, "Increasing students' awareness of their behavior in online learning environments with visualizations and achievement badges," *IEEE Transactions on Learning Technologies*, vol. 8, no. 3, pp. 261–273, 2015.

[135] P. Denny, A. Luxton-Reilly, M. Craig, and A. Petersen, "Improving complex task performance using a sequence of simple practice tasks," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 2018, pp. 4–9.

[136] J. Prather, B. A. Becker, M. Craig, P. Denny, D. Loksa, and L. Margulieux, "What do we think we think we are doing? metacognition and self-regulation in programming," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, 2020, pp. 2–13.

[137] L. E. Margulieux, M. Guzdial, and R. Catrambone, "Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 71–78. [Online]. Available: https://doi.org/10.1145/2361276.2361291

[138] R. P. Carver, "The case against statistical significance testing, revisited," *The Journal of Experimental Education*, vol. 61, no. 4, pp. 287–292, 1993.

[139] F. Dosseville, S. Laborde, and R. Lericollais, "Validation of a chronotype questionnaire including an amplitude dimension," *Chronobiology international*, vol. 30, no. 5, pp. 639–648, 2013.