CodeProcess Charts: Visualizing the Process of Writing Code

Raj Shrestha
Utah State University
Logan, Utah, USA
raj.shrestha86@outlook.com

Juho Leinonen
University of Helsinki
Helsinki, Finland
juho.leinonen@helsinki.fi

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Petri Ihantola
University of Helsinki
Helsinki, Finland
petri.ihantola@helsinki.fi

John Edwards
Utah State University
Logan, Utah, USA
john.edwards@usu.edu

ABSTRACT

Instructors of computer programming courses evaluate student progress on code submissions, exams, and other activities. The evaluation of code submissions is typically a summative assessment that gives very little insight into the process the student used when designing and writing the code. Thus, a tool that offers instructors a view into how students actually write their code could have broad impacts on assessment, intervention, instructional design, and plagiarism detection. In this article we propose an interactive software tool with a novel visualization that includes both static and dynamic views of the process that students take to complete computer programming assignments. We report results of an exploratory think-aloud study in which instructors offer thoughts as to the utility and potential of the tool. In the think-aloud study, we observed that the instructors easily identified multiple coding strategies (or the lack of thereof), were able to recognize plagiarism, and noticed a clear need for wider dissemination of tools for visualizing the programming process.

CCS CONCEPTS

• **Human-centered computing** → *Visualization systems and tools*;
• **Social and professional topics** → *Computing education.*

KEYWORDS

process data, visualization, software visualization, tool, visualization tool, source code analysis, source code snapshots, learning analytics, educational data mining

ACM Reference Format:
Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. 2022. *CodeProcess* Charts: Visualizing the Process of Writing Code. In *Australasian Computing Education Conference (ACE '22), February 14–18, 2022, Virtual Event, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511861.3511867

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 [38]. 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 [17, 34]. 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 [19], one example of which is keystroke data [23]. 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 [44]. Most previous work on such data has focused on analyzing compilations and predicting course outcomes [1, 4, 7, 8, 20, 24, 26, 46], understanding typing behaviors [11, 12, 25, 27, 33, 48], studying how novices construct programs [43], improving programming skill through code playback [10], 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 [29, 32], 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 [45]. 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

[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.

differencing functionality seen in some other programming process visualization tools (c.f. [13, 28]), 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 3, we present the *CodeProcess* tool. Section 4 presents the think-aloud study, and Section 5 gives conclusions.

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 [9]. In this study we talk about evolution of students' code. Evolutional visualizations are used for many purposes, including (professional) project management and understanding developers [29]. 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 more fine-grained data such as keystroke data [19]. The last decades have witnessed a noticeable increase in collection and use of snapshot data from introductory programming classrooms [6, 16, 19, 21, 22, 39].

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) [47] 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) [28] 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 [13] 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 [41] 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. [13] 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 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 [40], or in general by highlighting how the structure of the code changes over time. As a concrete example of the latter, Helminen et al. [15] and Piech et al. [35] 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. [36] 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.

2.3 Code measures over time

Some of the tools plot various aggregate statistics over time. For example, SCORE [47], PPV [28], Retina [30], and ClockIt [31] 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 [5]. Many of these metrics used in education (e.g., code complexity [18]) 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 [3] takes in tab-delimited data to build visualizations from students' programming process; similarly, ArAl [2] 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) [28], 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.

2.4 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* [45] 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" [45]. 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. 4.5) but also in interpreting the actions of the student (Sec. 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.

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.

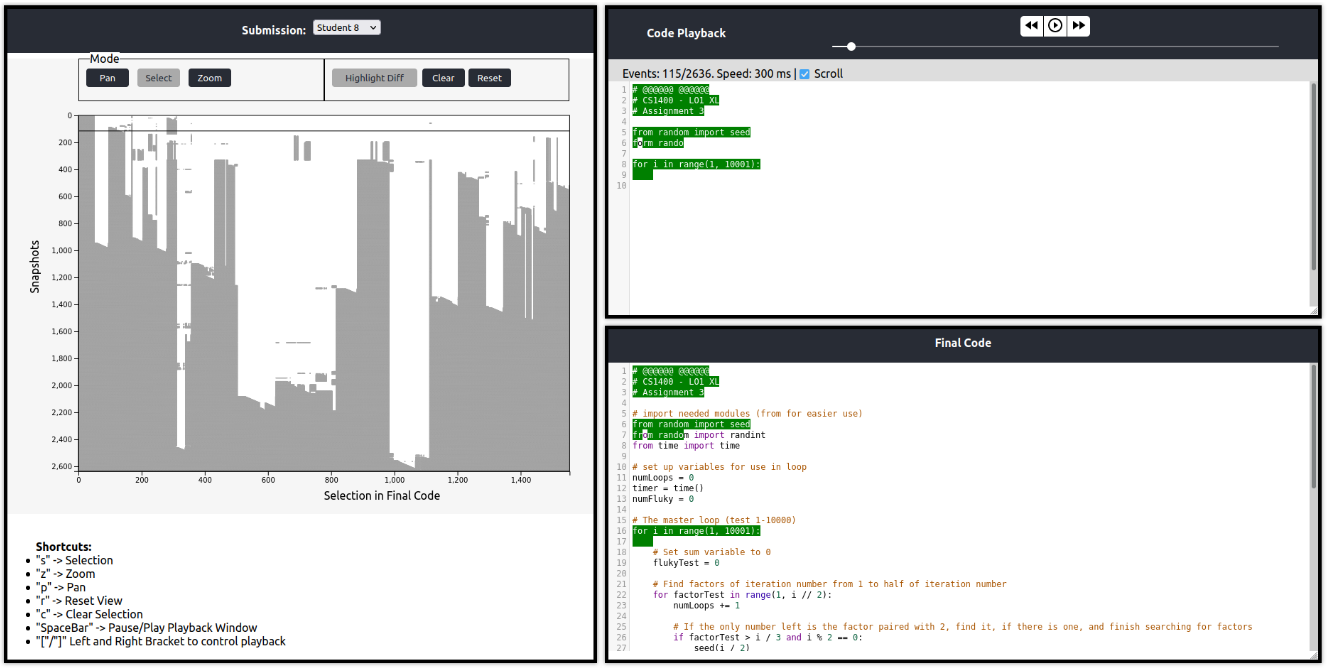
Given keystroke logs, we first preprocess them into a file that indexes the data. Any keystroke log can be converted to our format provided it has the inserted/deleted code, information about where in the code the change occured (either a single index into linearized code or row/column pair), and a timestamp. 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 1.

3.1 *CodeProcess* chart

The *CodeProcess* chart is the centerpiece of the software tool. See Figure 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 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 that the student typed correct code then deleted it.

In Figure 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 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 4 shows a zoomed in version of submission 11. The zoomed out version of submission 11 (Figure 2d) looks like a solid



Shortcuts:
- "s" -> Selection
- "z" -> Zoom
- "p" -> Pan
- "r" -> Reset View
- "c" -> Clear Selection
- "SpaceBar" -> Pause/Play Playback Window
- "["/"]" Left and Right Bracket to control playback

Events: 115/2636. Speed: 300 ms | ✓ Scroll

```
# @@@@@@ @@@@@@
# CS1400 - L01 XL
# Assignment 3

from random import seed
form rando

for i in range(1, 10001):

```

```
# @@@@@@ @@@@@@
# CS1400 - L01 XL
# Assignment 3

# import needed modules (from for easier use)
from random import seed
from random import randint
from time import time

# set up variables for use in loop
numLoops = 0
timer = time()
numFluky = 0

# The master loop (test 1-10000)
for i in range(1, 10001):

    # Set sum variable to 0
    flukyTest = 0

    # Find factors of iteration number from 1 to half of iteration number
    for factorTest in range(1, i // 2):
        numLoops += 1

        # If the only number left is the factor paired with 2, find it, if there is one, and finish searching for factors
        if factorTest > i / 3 and i % 2 == 0:
            seed(i / 2)
```

Figure 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.

block, indicating a large paste. Indeed, this is the case, which indicates a plagiarized submission. If we zoom in (Figure 4) we see thin vertical white lines. These lines indicate places where the student modified variable names and comments to mask the plagiarism.

Figure 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 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.

3.2 Code playback

The playback window (Figure 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 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.

3.3 Final code

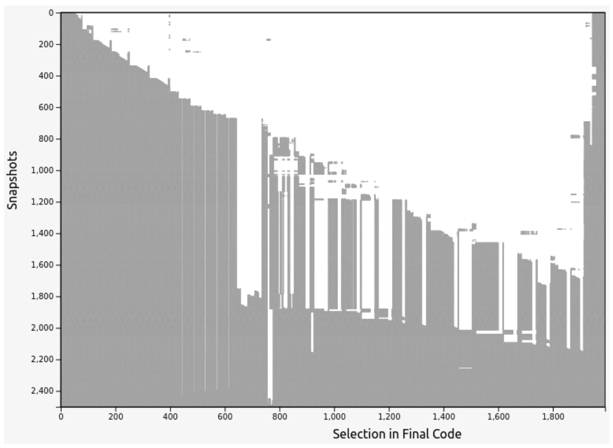
The final code window (Fig 1) displays the submitted solution of a student. If we highlight a particular section on the of the *CodeProcess* chart, it will highlight that section on the final code. The user can compare the final code with the snapshot and see the differences using the highlight feature.

4 THINK-ALOUD STUDY

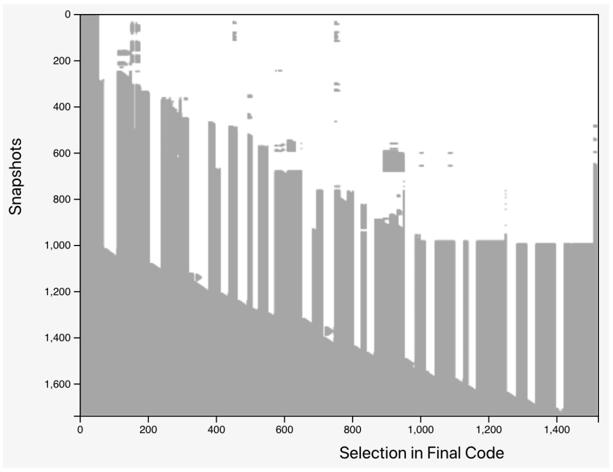
In this section, we provide the results of exploratory think-aloud sessions with two instructors. We discuss the different use cases of the tool along with instructors' thoughts and their experience with the tool.

4.1 Context and data

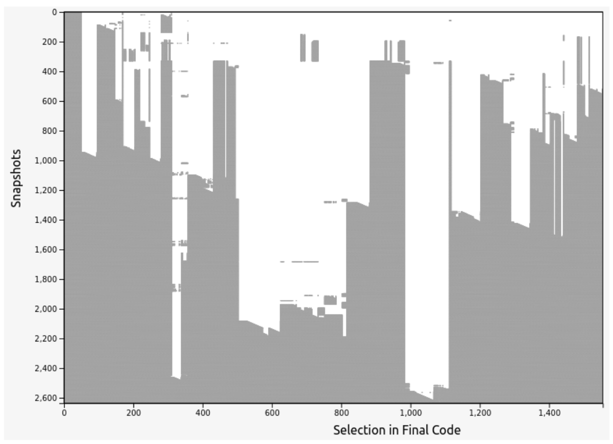
We collected keystrokes in a CS1 course during Spring semester, 2021. The course was taught online (because of COVID-19) at a mid-sized public university in the United States. At the beginning of the semester students were given the opportunity to opt into the study according to the university's IRB 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



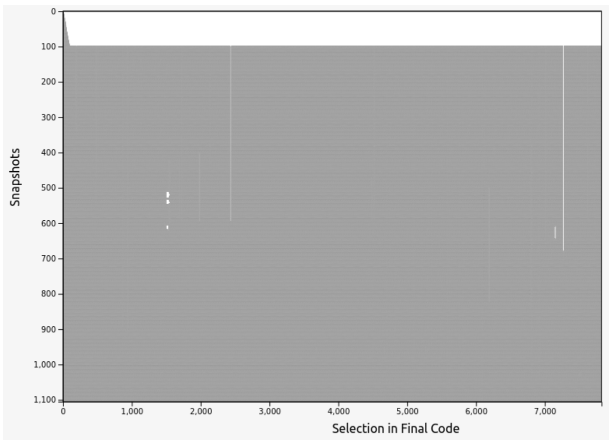
(a) Submission 4



(b) Submission 5



(c) Submission 8



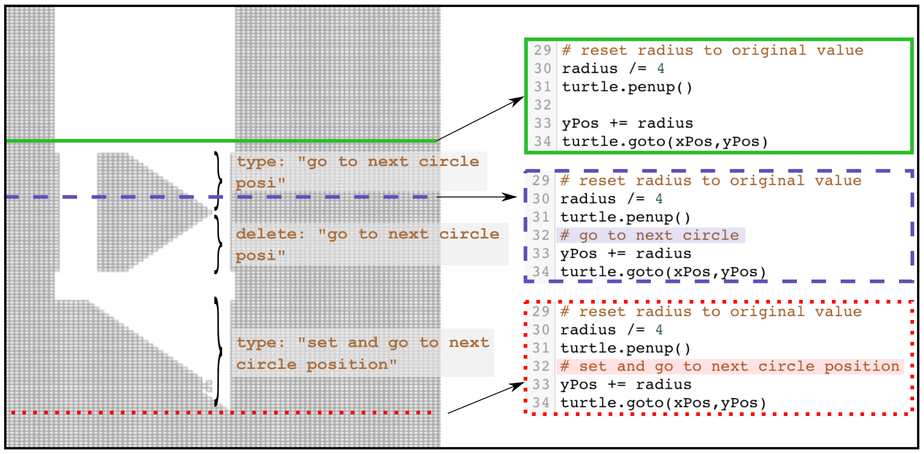
(d) Submission 11

Figure 2: Visualization of submissions 4, 5, 8, and 11. All but submission 5 were used in the pilot study.

acknowledge that their keystrokes would be recorded. The plugin recorded keystrokes while students wrote their programming assignments. A total of 81 submissions were collected.

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). 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.



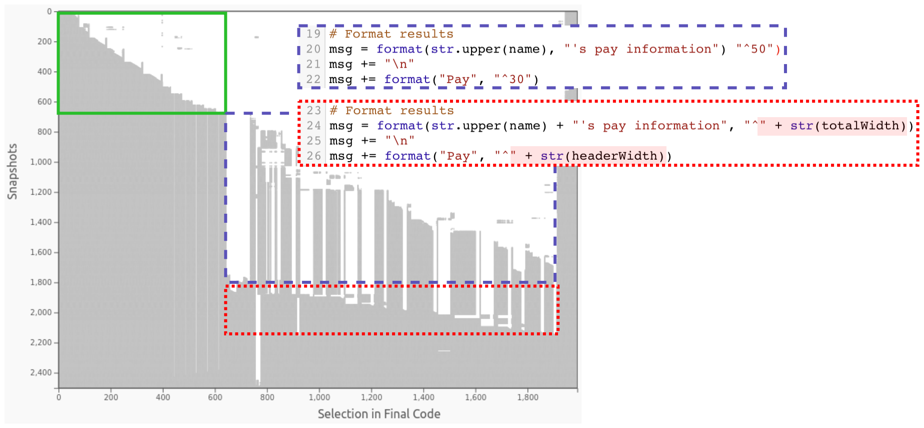
```
# reset radius to original value
radius /= 4
turtle.penup()

yPos += radius
turtle.goto(xPos,yPos)
```

```
# reset radius to original value
radius /= 4
turtle.penup()
# go to next circle
yPos += radius
turtle.goto(xPos,yPos)
```

```
# reset radius to original value
radius /= 4
turtle.penup()
# set and go to next circle position
yPos += radius
turtle.goto(xPos,yPos)
```

(a)



```
# Format results
msg = format(str.upper(name), "'s pay information") "^50")
msg += "\n"
msg += format("Pay", "^30")
```

```
# Format results
msg = format(str.upper(name) + "'s pay information", "^" + str(totalWidth))
msg += "\n"
msg += format("Pay", "^" + str(headerWidth))
```

(b)

Figure 3: (a) Zoomed in version of submission 5 (Figure 2b) at events 1307-1434 and including characters 705-770. (b) Details on submission 4 (Figure 2a).

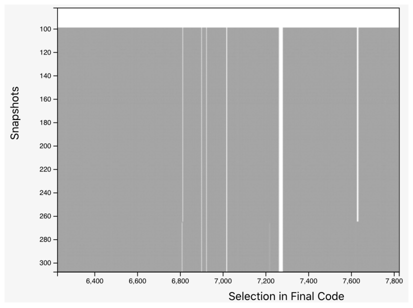


Figure 4: Zoomed in view of submission 11. See Fig. 2d.

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.

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

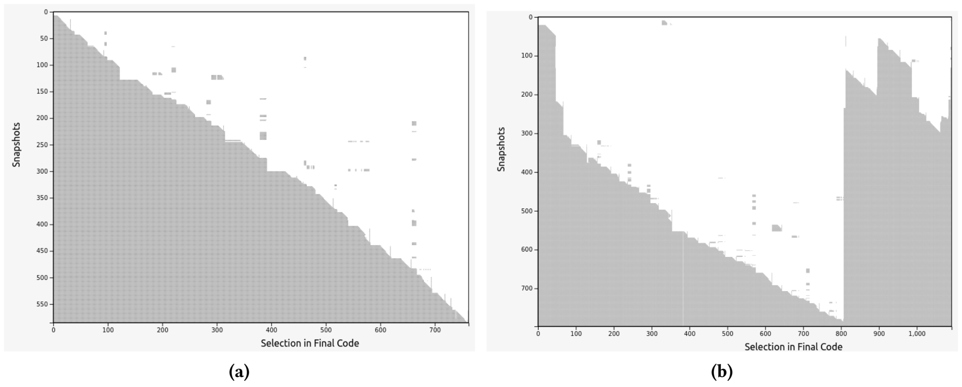


Figure 5: (a) Example of a student typing in code from another solution. (b) Example of what might look like a student typing code from another solution but what, in this case, is a capable student writing their code linearly.

the student edited a particular section of code. After looking at submission 4 (Figure 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 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 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.

4.4 Plagiarism detection

Detecting and proving plagiarism has always been a challenging problem. Plagiarism detection methods like MOSS [37] 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 [14]. *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 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 (who were not the

[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

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.

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 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.

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.

statement in the informed consent 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.

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.

5 CONCLUSIONS

In this work, we presented *CodeProcess* which is a novel tool for visualizing the programming process (example visualizations shown in Figure 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 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.

REFERENCES

[1] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. 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*. 121–130.

[2] Alireza Ahadi, Raymond Lister, and Luke Mathieson. 2019. ArAl: An Online Tool for Source Code Snapshot Metadata Analysis. In *Proceedings of the Twenty-First Australasian Computing Education Conference*. 118–125.

[3] Evan Balzuweit and Jaime Spacco. 2013. SnapViz: visualizing programming assignment snapshots. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 350–350.

[4] Brett A Becker. 2016. 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*. 296–301.

[5] Jacob T Biehl, Mary Czerwinski, Greg Smith, and George G Robertson. 2007. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1313–1322.

[6] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 223–228.

[7] Adam S Carter, Christopher D Hundhausen, and Olusola Adesope. 2015. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*. 141–150.

[8] Karo Castro-Wunsch, Alireza Ahadi, and Andrew Petersen. 2017. 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*. 111–116.

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

[10] Joseph Ditton, Hillary Swanson, and John Edwards. 2021. External Imagery in Computer Programming. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1226–1231.

[11] John Edwards, Juho Leinonen, Chetan Birthare, Albina Zavgorodniaia, and Arto Hellas. 2020. 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*. 204–215.

[12] John Edwards, Juho Leinonen, and Arto Hellas. 2020. A study of keystroke data in two contexts: Written language and programming language influence predictability of learning outcomes. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 413–419.

[13] Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. 2014. Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 229–234.

[14] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education*. 238–243.

[15] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. 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*. 119–126.

[16] Christopher David Hundhausen, Daniel M Olivares, and Adam S Carter. 2017. IDE-based learning analytics for computing education: a process model, critical review, and research agenda. *ACM Transactions on Computing Education (TOCE)* 17, 3 (2017), 1–26.

[17] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[18] Petri Ihantola and Andrew Petersen. 2019. Code complexity in introductory programming courses. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*.

[19] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. *Proceedings of the 2015 ITiCSE on Working Group Reports* (2015), 41–63.

[20] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*. 73–84.

[21] Philip M Johnson, Hongbing Kou, Joy M Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. 2004. 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, 136–144.

[22] Ayaan M Kazerouni, Stephen H Edwards, T Simin Hall, and Clifford A Shaffer. 2017. 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*. 104–109.

[23] Juho Leinonen. 2019. *Keystroke Data in Programming Courses*. Ph.D. Dissertation. University of Helsinki.

[24] Juho Leinonen, Leo Leppänen, Petri Ihantola, and Arto Hellas. 2017. Comparison of time metrics in programming. In *Proceedings of the 2017 acm conference on international computing education research*. 200–208.

[25] Juho Leinonen, Krista Longi, Arto Klami, Alireza Ahadi, and Arto Vihavainen. 2016. Typing patterns and authentication in practical programming exams. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 160–165.

[26] Juho Leinonen, Krista Longi, Arto Klami, and Arto Vihavainen. 2016. Automatic inference of programming performance and experience from typing patterns. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 132–137.

[27] Krista Longi, Juho Leinonen, Henrik Nygren, Joni Salmi, Arto Klami, and Arto Vihavainen. 2015. Identification of programmers from typing patterns. In *Proceedings of the 15th Koli Calling conference on computing education research*. 60–67.

[28] Yoshiaki Matsuzawa, Ken Okada, and Sanshiro Sakai. 2013. Programming process visualizer: a proposal of the tool for students to observe their programming process. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 46–51.

[29] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väätäjä. 2016. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*. 262–271.

[30] Christian Murphy, Gail Kaiser, Kristin Loveland, and Sahar Hasan. 2009. Retina: helping students and instructors based on observed programming activities. In *Proceedings of the 40th ACM technical symposium on Computer Science Education*. 178–182.

[31] Cindy Norris, Frank Barry, James B Fenwick Jr, Kathryn Reid, and Josh Rountree. 2008. 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*. 37–41.

[32] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. 2013. Software evolution visualization: A systematic mapping study. *Information and Software Technology* 55, 11 (2013), 1860–1883.

[33] Petrus Peltola, Vilma Kangas, Nea Pirttinen, Henrik Nygren, and Juho Leinonen. 2017. Identification based on typing patterns between programming and free text. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 163–167.

[34] Raymond Scott Pettit, John D Homer, Kayla Michelle McMurry, Nevan Simone, and Susan A Mengel. 2015. Are automated assessment tools helpful in programming courses?. In *2015 ASEE Annual Conference & Exposition*. 26–230.

[35] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*. 195–204.

[36] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 153–160.

[37] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 76–85.

[38] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 164–170.

[39] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin* 38, 3 (2006), 13–17.

[40] Alexandru Telea and David Auber. 2008. Code flows: Visualizing structural evolution of source code. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 831–838.

[41] Daniel Toll. 2016. *Measuring Programming Assignment Effort*. Ph.D. Dissertation. Faculty of Technology, Linnaeus University.

[42] Daniel Toll and Anna Wingkvist. 2018. Visualizing Programming Session Timelines. In *Proceedings of the 11th International Symposium on Visual Information Communication and Interaction*. 106–107.

[43] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. 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*. 109–116.

[44] Arto Vihavainen, Matti Luukkainen, and Petri Ihantola. 2014. Analysis of source code snapshot granularity levels. In *Proceedings of the 15th annual conference on*

information technology education. 21–26.

[45] Lucian Voinea, Alex Telea, and Jarke J Van Wijk. 2005. CVSscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization*. 47–56.

[46] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies*. IEEE, 319–323.

[47] Maximilian Rudolf Albrecht Wittmann, Matthew Bower, and Manolya Kavakli-Thorne. 2011. 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*. 118–122.

[48] Albina Zavgorodniaia, Raj Shrestha, Juho Leinonen, Arto Hellas, and John Edwards. 2021. 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, 261–272.