Eliph: Effective Visualization of Code History for Peer Assessment in Programming Education

Jungkook Park, Yeong Hoon Park, Suin Kim, Alice Oh
School of Computing, KAIST, Republic of Korea
{pjknkda, yhpark92, suin.kim}@kaist.ac.kr, alice.oh@kaist.edu

ABSTRACT
In this paper, we investigate the effectiveness of visualization of code history on peer assessment in computer science education. Peer assessment is found to be an effective learning tool for programming education. While many systems are proposed to support peer assessment in programming education, little effort has been devoted to finding ways to improve the peer assessment by assisting the students to understand the programs they are assessing. We introduce Eliph, a web-based peer assessment system for programming education with code history visualization. Eliph incorporates the visualization of character-level code history, selection-based history tracking and the integration of execution events to assist students in understanding programs written by peers, thereby leading to more effective peer assessment. We evaluate Eliph with an experiment in an undergraduate CS course. We show that visualization of code history has positive effects on promoting higher quality of peer feedback by understanding the intention and thought process.

Author Keywords
Peer assessment; Peer review; Data visualization; Time series visualization; Code history;

ACM Classification Keywords
K.3.1. Computer Uses in Education: Collaborative learning

INTRODUCTION
In education research, *peer assessment*, where students evaluate other students' assignments, is known to be an effective pedagogical tool. Studies found that peer assessment helps students develop higher-order cognitive skills by letting them engage in active analysis and judgement of other students' work [1]. In recent years, the use of peer assessment has rapidly grown because of massive open online courses (MOOCs) platforms that require a scalable solution for grading tens of thousands of assignments [6].

In computer science (CS) education, peer assessment involves students reviewing other students' code and giving marks and feedback, and several studies in CS education found positive effects of peer assessment. Studies found that peer assessment helps students improve their programming skills [21, 17] and delivers relevant and useful feedback to students [17]. Other studies found that peer assessment helps students to gain skills to evaluate technical work, understand their own progress better, and feel a sense of being part of a learning community [18, 4].

Peer assessment in CS education is most effective when it is well supported by tools. Some of the previous studies use commercial learning management systems or general peer review tools, most of which are not designed for code review [18]. Others use custom-built interfaces [17, 4]. However, these studies describe tools that support the process of peer assessment to run smoothly, rather than focusing on the efficacy of the tools in improving the quality of the feedback in peer assessment. For example, [4] introduces code annotation tool that let students give more detailed feedback on specific parts of the code during the code review.

Our approach to assist students in peer assessment is to give them rich information about the whole problem solving process of a given code, such that they can understand the intentions of the author of the code. This is motivated by research that even for skilled programmers, it is difficult to infer the intentions of the author by merely reading the code [7], and one of the common practices to overcome this is to use a system for browsing the history of the code.

Thus, we explore the idea of showing the code history to students participating in peer assessment. In addition to improved results of peer assessment in programming education, we conjecture that showing the code history has more benefits. For instance, by observing the complete history of the code writing process of others, students may gain more knowledge about coding styles, steps of designing the structure of a program, and other programming skills that may be unfamiliar to them. Also, these benefits may lead to improvement in the reliability of peer assessment. Although there are studies about finding a good aggregation algorithm for more accurate grading [13, 14, 16], recent findings suggest that these methods cannot make significant improvements because the errors originate from the lack of information or the misunderstanding of the topic [16]. We hypothesize that the added information of code history will positively affect the accuracy of peer grading, resulting in improved reliability (or reduced variance).

In this paper, we investigate the effectiveness of the visualization of code history for peer assessment in programming education. We evaluate three hypotheses: 1) browsing the code history results in improved quality of peer feedback; 2) browsing the code history helps students get positive learning outcomes during peer assessment; 3) browsing the code history improves the reliability of peer assessment.

While there are several tools for browsing code history, we found that the existing systems are optimized for software development and not well suited for peer assessment in programming education. We present **Eliph**, a web-based peer assessment system for programming education with code history visualization. Eliph incorporates three major features to promote effective peer assessment in programming education: (i) *Character-level code history*, (ii) *Selection-based history tracking*, and (iii) the integration of *Execution events*.

In the rest of the paper, we describe Eliph, a peer assessment system with a visualization tool for character-level code history and selection-based code history. We explain how we integrate Eliph into an online CS education platform to conduct an experiment in peer assessment. We then present the results of the experiment that show Eliph is in fact effective in improving the quality of peer assessment.

RELATED WORK

In this section, we describe related work in two research topics, peer assessment in programming education and code history visualization, and explain how our research contributes to these topics.

Peer Assessment in Programming Education

Peer assessment is known as an effective pedagogical tool and has been widely adopted in tertiary education across a variety of disciplines [20, 2, 1]. There are also studies about its effectiveness when it is introduced as a form of peer code review in programming education [9]. Here we describe some of the studies that propose systems for supporting peer code assessment. [17] is one of the early studies, that describes a web-based system for peer code review and assessment, showing preliminary findings on the effectiveness of peer assessment in learning programming languages. [4] introduces a Web-based Programming Assisted System (WPAS) with annotation tool where students are able to highlight and give comments on parts of the code. Whereas these systems only show the final version of the code, we incorporate the code history to augment the peer assessment and show the effects through a classroom experiment.

Visualization of Code History

There are several tools and papers about visualization of code history. A survey paper [11] describes previously presented techniques to record and utilize developers' operations on the integrated development environment (IDE). *SpyWare* [15] records code changes by hooking IDE's change notification. *Syde* [3] records changing history of abstract syntax tree (AST). However, these two methods are based on the syntax-element-level change history such as AST and have a strong dependency on the specific programming language. Also, extracting of syntax elements from the code requires an AST parser, therefore cannot parse incomplete code which has any syntactic error. To overcome the limitation of syntax-element-level change history, *OperationRecorder* [12] increases the granularity of data by extracting all editing operations from the undo history of Eclipse IDE. Closely related to our research is *AZURITE* [23], an Eclipse IDE plugin which collects character-level code change history and visualizes it. While Azurite is developed and tested for increasing productivity in software engineering, we specially developed and integrated the code history visualization into the education setting for improving peer assessment.

ELIPH

In this section, we describe the detailed elements of our code peer assessment tool Eliph. First, the most important capability of Eliph is that it shows a *character-level code history*, rather than the differences between snapshots. Second, Eliph lets the user select any arbitrary block of the code for *selection-based history tracking*, as viewing fine-grained history may be cumbersome, in cases where the code is very long. Third, another element of Eliph to assist students in peer assessment is the visualization of *execution events* that include system output and error messages. Lastly, we combine all of these elements into a *student interface* that consists of the code viewer, the history viewer, and the history slider. Figure 1 shows a screenshot of this interface. Below we explain each of the elements in detail.

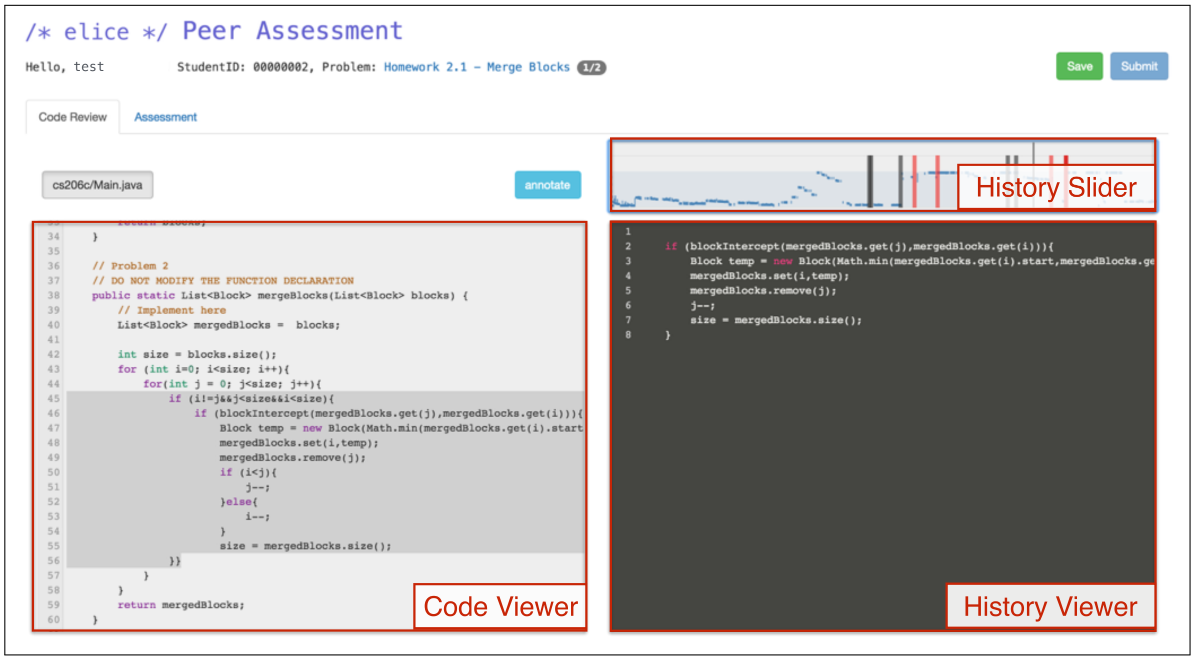
Character-Level Code History

For effective visualization of code history, Eliph keeps the code change history at a fine-grain, as small as one character insertion and deletion. This is in agreement with recent research that it is important to look at code evolution at a much finer grain than in traditional version control systems (VCS) [10, 15]. Operational transformation (OT) is one effective approach for working with real-time document changes at a fine grain [19], and we use it to efficiently collect, store, and re-create code revision history. OT works by identifying the operations between two versions of a document, for example *inserting* a word and *deleting* a character. To identify the position of the insertion and deletion, a *skip* operation is used. Figure 2 shows an example of the application of these operational transformations.

Eliph is integrated with the code editor provided by the Elice platform [5], a web-based computer science education platform. When a student works on a programming exercise on the Elice platform, it logs all the code changes at a character level. Eliph is provided with the code changes data, then it processes the data as OTs and uses them to re-create and visualize the code history.

Selection-Based History Tracking

For some programming exercises, the history of the entire code can be too vast and complex to navigate, and as such, the visualization tool should provide appropriate filtering and aggregation to organize and summarize the fine-grained history data. Our approach for filtering and aggregation of the



```
    }

    // Problem 2
    // DO NOT MODIFY THE FUNCTION DECLARATION
    public static List<Block> mergeBlocks(List<Block> blocks) {
        // Implement here
        List<Block> mergedBlocks =  blocks;

        int size = blocks.size();
        for (int i=0; i<size; i++){
            for(int j = 0; j<size; j++){
                if (i!=j&&j<size&&i<size){
                    if (blockIntercept(mergedBlocks.get(j),mergedBlocks.get(i))){
                        Block temp = new Block(Math.min(mergedBlocks.get(i).start
                        mergedBlocks.set(i,temp);
                        mergedBlocks.remove(j);
                        if (i<j){
                            j--;
                        }else{
                            i--;
                        }
                        size = mergedBlocks.size();
                }}
            }
        }
        return mergedBlocks;
    }
```

```

    if (blockIntercept(mergedBlocks.get(j),mergedBlocks.get(i))){
        Block temp = new Block(Math.min(mergedBlocks.get(i).start,mergedBlocks.ge
        mergedBlocks.set(i,temp);
        mergedBlocks.remove(j);
        j--;
        size = mergedBlocks.size();
    }
```

Figure 1. Overview of the Eliph interface. The code viewer (left) shows the last version of the code. The history slider (upper right) shows execution events and the relative location of code change through the code history of selected code block. User can drag the history slider horizontally in order to browse the code history. The history viewer (right) shows the code corresponding to the version indicated by the history slider.

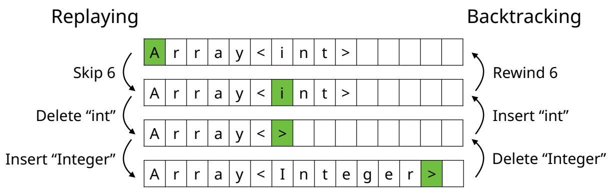


Figure 2. Example usage of Eliph's OT for the character-level code history. Green box indicates the position of the current cursor.

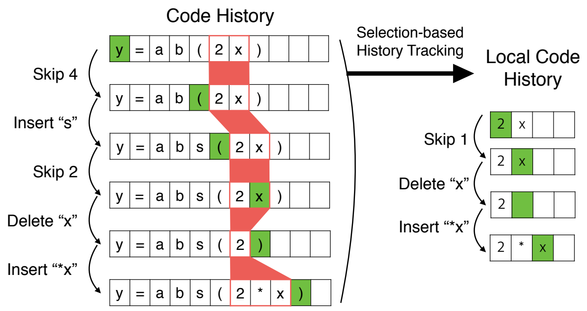


Figure 3. Example usage of Eliph's selection-based history tracking. The green box indicates the position of the current cursor. The red region indicates the area of the selected code block. The entire code history is composed of five OTs, and after user selects the argument part of the function call, the resulting local code history contains three OTs that are relevant to the selected code block.

fine-grained code history is to provide a selection-based history tracking feature. That feature allows the user to select any arbitrary block of the code, then Eliph extracts and visualizes the code history of the selected code block. Figure 3 shows one example of selection-based history tracking. In the example, there are different parts in the assignment statement that has been changed, but when the user selects the argument part of the function call, Eliph tracks all relevant OTs and reconstructs a local code history that only contains the OTs that are relevant to the selected region.

Execution Events

In Elice, students repeat the cycle of code writing, submission, compilation, execution and grading to solve programming exercises. We define execution events as the set of all events generated from the cycle. The standard output generated from the code execution is one type of execution event, and it can reveal the author's intention because standard output is often used for debugging programming codes. Likewise, error logs from compilation and execution process are also regarded as execution events. Automated grading result, another execution event, usually consists of scores and messages from pre-defined test cases written by instructor. These execution events are recorded through automated grading script in the Elice platform, which compiles, executes, and grades student's code submission.

Eliph visualizes the execution events to let the users refer them as clues to find out what the code author was trying to achieve at the moment. For example, by observing several execution events that contain error messages within a small time window, the user may infer that the code author was struggling with debugging the code and fixing an error during that time.

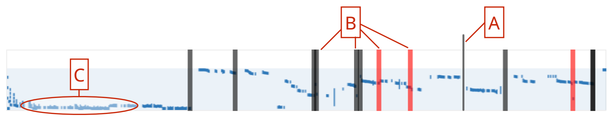


Figure 4. History slider in Eliph. The slider visualizes the timeline of code history. A: The current position of the slider, draggable with pointing device. B: Execution events; red and black color represents execution events containing error message and non-error output, respectively. C: Vertical position of the blue dots represent the location of the code change within the file.

Student Interface

The Eliph interface integrates the visualization of character-level code history, selection-based code history, and execution events described above. The interface consists of three components (Figure 1): *Code Viewer*, *History Slider*, and *History Viewer*. Students may look through the last version of the code in the *Code Viewer*, and select a region of code to browse the code history of the selected block. As the student makes a selection, *History Slider* visualizes the timeline of the code history and execution events relevant to the selected code block. Then, the student can click and drag over the *History Slider* to browse through different versions of the selected code block in the *History Viewer*.

The design of the *History Slider* is shown in Figure 4 in detail. The horizontal axis represents the revisions of the code, and the thin vertical line (A) shows the current position of the slider. As a student drags the slider, the line follows the mouse pointer and the corresponding version of the code is displayed in *History Viewer*. The vertical bars (B) represent the execution events. The red colored bars represent error messages, for example, compilation or runtime error. The student can click one of these bars to see the details of the execution event. The blue shapes (C) represent the relative location of the code change within the file, visualizing where the code changes have occurred over time.

EVALUATION

We conducted an experiment with the students in an undergraduate computer science course (Data Structures) to investigate the benefits of showing code history in peer assessment in programming education. We designed the experiment to examine the following three hypotheses:

- H1: Visualization of code history promotes higher quality of peer feedback.
- H2: Visualization of code history helps student to get positive learning outcomes.
- H3: Visualization of code history improves the reliability of peer assessment.

In the *Introduction to Data Structures* course, the majority of students (66%) were in their second year, and the second largest group of students are in their first year (18%). The course used Java as a primary programming language.

At the beginning of the course, we conducted a *pre-course survey* to ask students how they feel about the course. We used questions designed for the expectancy–value theory [22]

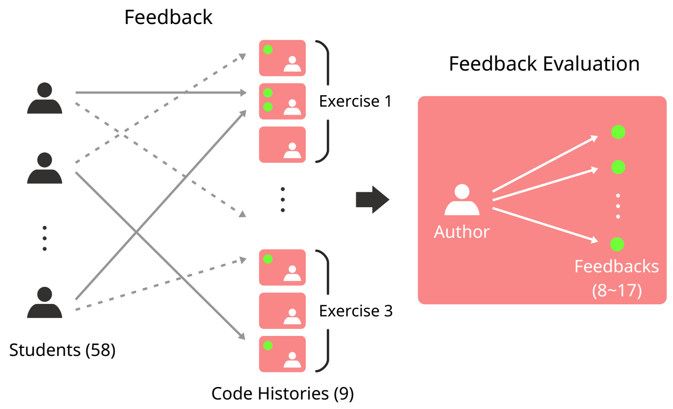


Figure 5. Overview of the two-step experiment. We randomly selected nine code submissions from three programming exercises. Step 1 (left): Each of the 58 students was asked to evaluate two code submissions, once using code history (solid line), and once without code history (dashed line). Step 2 (right): Code authors were asked to evaluate the feedback they received. 7 out of 9 authors participated in this evaluation, and the number of the feedback each author received ranges from 8 to 17.

to measure (i) A1: how good students believe they already are for the given subject (ability belief items); (ii) A2: how well students expect to do in the course (expectancy items); and (iii) A3: how important and interesting students think the course is (usefulness, importance, and interest items). Students responded with medium-low belief in their current ability (A1: $\mu = 2.70, \sigma = 1.04$, from 5-scale Likert question), medium-high expectancy of their performance (A2: $\mu = 3.81, \sigma = 0.90$), and very high interest on the subject (A3: $\mu = 4.49, \sigma = 0.69$). In summary, the result indicates that most students in this class think the course is very important and are willing to study most of the course material, but they think their current abilities are below average. Most notably, almost all students responded positively on questions of A2 and A3, indicating that students were highly motivated before class started.

We then designed a two-step experiment for students to validate the benefits of history visualization during the peer assessment. In particular, we sought to understand how Eliph helps the process of (i) students assessing their peer's work, and (ii) students being evaluated by peers.

Step 1: Feedback Generation

In this step, students participated in peer assessment during a regular class time. We first chose three programming exercises from homework problems, which all students were required to submit prior to our peer assessment experiment. For each of the three programming exercises, we randomly selected three representative students, resulting in nine code submissions. Then, we asked students to access the Eliph web page using their own laptop to start peer assessment. Each student was required to evaluate two anonymized code submissions among the nine code submissions, once with the code history and once without. Two of the code submissions were arbitrarily assigned to each student in random order to eliminate any potential ordering bias. Students were given a grading rubric for evaluation. Using the criteria, students

	Avg. score	% solved	Avg. # of executions
E_1	93.06	97% (91/93)	61.0
E_2	72.02	85% (79/93)	30.0
E_3	72.25	80% (74/93)	38.9

Table 1. Student statistics of the three programming exercises used in the case study. The score of each exercise ranges from 0 to 100. Students who got a score of 100 are marked as "solved".

were asked to score the given code, write comments, and/or directly add annotations to the code to "help the students being evaluated".

Programming Exercises

For each homework assignment, students were given two weeks to solve the exercises. Each programming exercise contains the problem description, the skeleton code, and a test/submission environment. Students were able to code, test run, and submit inside the web IDE in the Elice platform. There was no limit on the number of code submissions, thus students were able to submit their code any time to immediately view the score of their submission. Students "pass" the exercise when their submission passes all test cases and get the score of "100".

To thoroughly validate the effect of viewing code history, we chose three exercises at different levels of difficulty and length of code required to solve. Table 1 shows the student statistics for the three exercises. The first exercise (E_1) has the lowest level of difficulty among three, and 97% (91 out of 93) of students passed the exercise. Second (E_2) and third (E_3) exercises have higher level of difficulty than E_1, resulting in only 85% and 80% of students who passed the exercise, respectively. Homework problem E_3 required students to write longer code compared to the others.

Code History Selection

For each programming exercise, we randomly selected three code submissions. Since we allowed students to code outside of the Elice platform, there were some students who wrote their code and copy-pasted from outside of the platform. We excluded these students because we need submissions with complete code history from the initial skeleton code to the final submission. We let $c_{i,j}$ denote code submission j for programming exercise i. We chose nine code submissions ($c_{1,1}$ to $c_{3,3}$), each from nine distinct students.

Peer Evaluation

We conducted peer evaluation during a regular class time. Students brought their own laptops and agreed on the consent form to participate, with a note that they can stop participating at any time. Additionally, we instructed that the participation of this test would not affect their grade in the class. 58 out of 93 students agreed to participate in the peer evaluation experiment. At the beginning of the experiment, the participating students were given a quick tutorial and a demonstration of how to use the features of Eliph.

Figure 5 shows the overview of the experiment design. Before the experiment, we randomly, but evenly assigned each code from $c_{1,1}$ to $c_{3,3}$ to each experiment participant. We ask

	Number of Student Feedback		
Code	w/ History	w/o History	Total
$c_{1,1}$	7	9	16
$c_{1,2}$	7	7	14
$c_{1,3}$	3	8	11
$c_{2,1}$	9	8	17
$c_{2,2}$	8	5	13
$c_{2,3}$	3	5	8
$c_{3,1}$	6	6	12
$c_{3,2}$	7	4	11
$c_{3,3}$	8	6	14
	58	58	116

Table 2. The number of feedbacks we collected on each code submission.

participant to evaluate two code submissions with two different settings, "with code history" and "without code history". In the "with code history" setting, participants may select a code block from the *Code Viewer*, to browse the history of the selected code block using the *Local History Viewer*. *History Slider* and *Local History Viewer* are hidden from the students for the "without code history" setting. We randomized the order of the programming exercises and code history settings (with/without code history) to avoid any potential ordering bias.

It is important to set up the criteria for peer code evaluation that is relevant and can be understood by students. We created the criteria based on other peer assessment studies [4, 17] and a grading rubric for an introductory computer science course [8], as follows:

- Correctness – Program has no errors; program always works correctly and meets the problem specification.
- Program Design – Program is well structured and easy to follow.
- Efficiency – The programmer made good decisions about the algorithms and language features for efficiency.
- Readability – Code is clean, understandable, and well-organized.
- Assignment Specifications – Details of the assignment specification are correctly followed.

To make sure students clearly understand all the criteria, we provided the students with a supplementary guide that describes each criterion in detail.

Students were asked to evaluate the code in three ways: (i) assign numerical score for each criterion (ii) write comments for each evaluation criterion, and (iii) directly annotate the final code snippet (shown in *Code Viewer*) by selecting the code block and pressing "annotate" button. The average assessment time for each code was 12.63 minutes.

No.	Question	Pos.(%) / Neg.(%)	Mean	SD
Q1	To understand how the code works, browsing the code history was more helpful than viewing the last version of the code.	39.66 / 18.97	3.19	0.96
Q2	To understand the code quickly, browsing the code history was more helpful than viewing the last version of the code.	36.21 / 24.14	3.14	1.07
Q3	To understand author's intention of the code, browsing the code history was more helpful than viewing the last version of the code.	68.97 / 13.79	3.86	1.06
Q4	To assess the code, browsing the code history was more helpful than viewing the last version of the code.	39.66 / 22.41	3.21	0.94
Q5	To provide feedback for the code, browsing the code history was more helpful than viewing the last version of the code.	39.66 / 18.97	3.24	0.97
Q6	To learn how to write correct code, browsing the code history was more helpful than viewing the last version of the code.	41.38 / 22.41	3.22	0.89
Q7	To learn how to write well-organized and readable code, browsing the code history was more helpful than viewing the last version of the code.	36.21 / 22.41	3.19	0.96
Q8	To learn how to write efficient code, browsing the code history was more helpful than viewing the last version of the code.	29.31 / 24.14	3.09	0.92

Table 3. The results of the post-evaluation survey. n = 58, and each question uses the 5-point likert scale. Positive percentages indicate the proportion of "agree" and "strongly agree", and the negative percentages indicate the proportion "disagree" and "strongly disagree".

Post-Evaluation Survey

After completing the peer assessment, the evaluators were asked how they felt about using code history during the assessment process. To evaluate the pure benefit of using code history, we asked students to compare between "with code history" and "without code history" settings. Specifically, we asked students how much they were able to understand the code and code author's intention, and how much they have learned during the assessment process in the "with code history" setting, compared to "without code history" setting. Questions were given with a 5-point Likert scale, from "Strongly Disagree" to "Strongly Agree". The questions and the student's responses are shown in Table 3. There were also two free-form questions in the survey, which are related to hypotheses H1 and H2 respectively.

Step 2: Feedback Evaluation

In this part, we recruited the authors of the nine code submissions and asked them to evaluate the feedback they received from the peer assessment. Seven authors consented to participating in the feedback evaluation, and two authors of $c_{2,3}$ and $c_{3,1}$ did not participate. For the evaluation, we provided a web-based feedback evaluation environment to the authors. The feedback evaluation session took place for about an hour in an offline classroom, with students participating on their own laptop computers.

Eliph's feedback evaluation page consists of a code viewer and an assessment result viewer. When the author of the code selects a feedback he/she received, the code viewer shows the code annotations from the peers by coloring the annotated area as bright-green and the assessment result viewer shows points and comments he/she got on each criterion in the grading rubric. To see the messages in code annotations, the author can move his/her mouse pointer over the bright-green-colored code line in the code viewer. All feedback comments were anonymized, and authors could not see whether the feedback had been generated with code history or without.

We asked the authors to fill out an evaluation form for each feedback. According to Table 2, every participating author was required to fill out 11 to 17 evaluation sheets. There were five criteria in the evaluation form and the responses were formed as 5-point Likert scale, from "Strongly Disagree" to "Strongly Agree". The criteria for evaluation and the authors' responses are shown in Table 4.

ANALYSIS

In this section, we present our analysis of the data from the two-step experiment with respect to three hypotheses. For H1 and H2, we employed a hybrid method of quantitative and qualitative analysis on three different sources of data: student feedback, post-evaluation student survey, and code authors' feedback evaluation. For H3, we quantitatively investigated the feedback evaluation data.

Data Preparation

We prepared the data by filtering out incomplete results and aggregating the free-form comments in the post-evaluation survey.

There are two cases of incomplete results we filtered out. First, some participants dropped out before completing the last step of the survey questionnaire. We regarded the feedback written by these users as "incomplete" and removed them from the data. Second, some students just clicked the submit button without writing any comments, and these were also regarded as "incomplete" and removed from the data.

We aggregated the free-form comments in the student post-evaluation survey after the feedback generation step. There were two free-form questions in the the survey regarding hypotheses H1 and H2 for which we obtained 57 and 53 responses, respectively. Two authors independently grouped the comments according to the main idea of the comment, then discussed to reach a consensus on the groupings.

We did not collect log data that are directly liked to the user behavior with the tool. However, after analyzing the responses from the post-evaluation student survey, we found that almost all the student (57 out of 58) who completed the survey made a solid opinion on whether it helped, and the reason why it helped (or not helped) in their assessment/learning.

H1: Eliph promotes higher quality of peer feedback

Quantitative Findings

To analyze the effect of showing the code history for promoting better quality of peer feedback, we divided all feedback into two groups – one is the experiment group with code history (Eliph) and the other is the control group without code history. Table 4 shows the result the code authors' evaluation of the feedback they received, with the averages and standard deviations of the scores on each criterion. The results show that the Eliph group (with code history) received significantly higher scores, especially for the criteria of "style" and "efficiency" of the code. Also, we find that the satisfaction about the quality of feedback is significantly higher in the Eliph group.

Furthermore, Table 3 shows that students felt more positive about giving feedback with the code history compared to just viewing the last version of the code. All questions including "understand the code" (Q1, Q2), "understand the author's intention" (Q3), "assess the code" (Q4) and "provide feedback" (Q5) have mean scores of greater than 3, and more positive responses than negative responses.

Qualitative Findings

57 students replied to the free-form question *"How did browsing the code history help you assess the code? If it did not, why?"*. In the responses, students generally agreed that browsing the code history helped them with assessing others code. However, their responses varied in how the code history helped with the assessment.

Intention: Nine students mentioned that they could infer the code author's intention from the code history.

> *"It allowed me to understand how the author came up with the particular algorithm as well as why he implemented some of the functions." (Student 13)*

> *"I could see in what order the author wrote the code. Thus, I was able to know the author's intention, and it was helpful for assessing the efficiency of the code." (Student 20)*

Thought Process: Eight students said they were able to follow the author's thought process by browsing the code history.

> *"Browsing the code history was helpful in understanding the author's flow of thought." (Student 23)*

> *"It felt like reading the steps of the thinking process." (Student 29)*

Trial-and-Error: Eight students reported that the code history showed the process of trial and error in the code history.

> *"The code history showed me the process of making mistakes and fixing them, so I was able to understand where the author had been mistaken." (Student 4)*

> *"It helped me understand how the author wrote the code, and how he/she had made progress with trial and error." (Student 53)*

Code Readability: Also, six students reported that they were able to understand the code more easily by browsing the code history.

> *"In cases of code with poor readability, I had to browse its code history to figure out the intention behind it." (Student 58)*

> *"... What helped the most was that I didn't have to understand the entire code at once; with the code history, it was easier because I could follow the author's stream of thought." (Student 57)*

On the other hand, some students argued that browsing the code history was not much helpful for assessing peer's code.

Unnecessary Information: The most frequently mentioned reason was that they felt the last version of the code was sufficient for the assessment.

> *"Since it wasn't a big project, I couldn't get much extra information out of the code history." (Student 2)*

> *"It did not help too much because the code was easy to understand." (Student 56)*

Confusion: Four students reported that they were confused by seeing the code history.

> *"No, I think code history is something that should be hidden. Looking at that would make it more difficult to understand because it confuses you with the program structure of the final version." (Student 17)*

H2: Eliph helps student to get positive learning outcomes

Quantitative Findings

The result of Q6, Q7 and Q8 in Table 3 shows that student assessors get positive learning outcome from using Eliph. For the question to "learn how to write correct code" (Q6), 41.37% of students agreed with the effect and showed the average score of 3.22 in the 5-point likert scale. For the question "learn how to write well-organized and readable code" (Q7), 36.21% of student agreed with the average score of 3.19. However, for the learning effect in "writing efficient code"(Q8), the result was neutral, with 29.31% agreeing and 24.14% disagreeing.

Qualitative Findings

53 students replied to the free-form question, *"How did browsing the code history help you learn to write a good code? If it did not, why?"*. On this question, the responses

No.	Criterion	Exp. Group	Control Group
R1	The peer clearly understood my code.	3.97 (0.90)	3.79 (0.74)
R2	The feedback will help me to improve the style or readability of my future code.	*3.72 (1.07)	3.24 (0.97)
R3	The feedback will help me to improve the efficiency or to use a better algorithm for my future code.	†3.72 (1.17)	3.21 (1.04)
R4	I feel the feedback is fair and unbiased.	3.81 (1.13)	3.55 (1.05)
R5	I am satisfied with the overall quality of the feedback.	*3.89 (0.94)	3.38 (0.95)

Table 4. The statistics of code authors' responses about the feedback from peer assessment. $^*p < 0.05$, † marginally significant. Seven code authors participated, and the number of feedback in the experiment group is 36, and control group 42.

were evenly divided, in accordance with the quantitative result; 21 students agreed that they learned from browsing code history, while an equal number of others disagreed, and the others neither agreed or disagreed.

Comments from 21 students reported that students got positive learning outcomes from browsing the code history.

Writing Readable Code: Five students said that they learned how to write a readable code from the peer's code history.

> *"I learned some techniques such as naming variables, assigning types for variables, splitting code into small pieces, which could prevent potential problems as the code gets bigger." (Student 14)*

> *"I learned to write a more readable code by watching the author's numerous efforts to make code more clean." (Student 34)*

Different Code Styles: Some students reported that browsing the code history helped them learn different ways of coding.

> *"Watching a different style of code writing, I feel like I came to realize the correct way to write code." (Student 38)*

> *"I realized that people write code using steps in different order. I learned more from code written by someone who codes more like myself." (Student 48)*

Trial-and-Error: Students can see the trial and error through the code history, which helped the students to learn how to overcome errors in specific situations.

> *"I learned how people deal with different situations and how people write simple and clean code. Most of all, watching the trial and error gave me insights into particular cases where some approaches simply don't work." (Student 33)*

Thought Process: Some students also mentioned how reading the author's thought process through the code history helped them to learn to code better.

> *"By knowing how the code was written and the thought process behind it, I was able to learn how to expand my thought." (Student 7)*

> *"It helped because I could figure out the thought process of the author, through which he/she wrote the code." (Student 41)*

On the other hand, about 40% students (21 of 53) described why they felt browsing the code history was not helpful.

Not Much to Learn: The most common reason was that they could not find many aspects to learn from the code history because the final version of the code would be more correct and optimized than any previous versions in many cases.

> *"Since students usually save the most efficient code in the last version, it seems to be helpful to see only the last version of the code." (Student 49)*

> *"If a well-written code is given, I could see the process of writing good code by looking only at the final version of the code" (Student 51)*

> *"It is easy to see from the last version of the code the parts which are not good." (Student 56)*

Poorly Written Code: Some students mentioned that they could not learn from their peer's code because the code was not good, or the code history contained wrong or inefficient code.

> *"Unless peer's code is perfect, viewing the code history does not seem to result in learning anything." (Student 16)*

> *"Code history is not helpful for learning something new because it contains wrong or inefficient code." (Student 44)*

H3: Eliph improves the reliability of peer assessment

To test H3, we compared the peer evaluation scores between the experiment group and the control group. We used the variance of scores as the metric of reliability of peer assessment. The average value and the standard deviation value of the scores on each evaluation criterion are listed in Table 5. We collected a total of 90 sets of peer evaluation, and the experiment group and the control group received 43 and 47 sets of feedback, respectively. To compare the score variance between the two groups, we conducted the Levene's test. The result shows there is no criterion with significant difference in the variance of the score between the two groups, which

	Avg. score (SD)		P-value	
	Exp. Group	Control Group	T-Test	Levene-Test
PD	18.42 (2.14)	17.87 (2.59)	0.282	0.286
E	16.37 (2.97)	16.68 (3.72)	0.667	0.539
R	13.33 (2.08)	12.70 (2.78)	0.234	0.494
AS	13.81 (2.97)	14.36 (1.85)	0.309	0.298
$\sum$	61.93 (7.45)	61.62 (7.64)	0.846	0.710

Table 5. The results of peer evaluation. Each criterion means PD=Program Design, E=Efficiency, R=Readability, AS=Assignment Specifications. For all evaluation criteria, both the mean and the variance are not significantly different between the two groups. The number of feedback in experiment group and control group is 43 and 47, respectively.

means that the reliability of peer assessment in the two groups does not differ significantly.

Additionally, we tested H3 with another reliability metric, defined as the average score of R4 in feedback evaluation. The results are shown in Table 4. Criterion R4 is about "fairness" and "unbiasedness" of the peer evaluation feedback. As shown in the result, the average score of R4 in the experiment group is higher than the control group, but the difference is not statistically significant.

DISCUSSION

Different experience-level of the students

The differences in experience levels may yield different implications for individual students with respect to the effectiveness of Eliph. In order to identify the differences from our result, we calculated the correlations between the data of the Ability Belief items in the pre-course survey (A1) and two averaged items from the post-evaluation survey result (Q1-5 is the average of questions from Q1 to Q5; and Q6-8 is the average of questions from Q6 to Q8) for every individual, each representing respectively the speculated measurements of student's experience level and his/her perceived effectiveness of Eliph. From a Pearson correlation analysis, no measurable correlation was found ($r = -0.152$ and $r = 0.172$ respectively). From the result, we conclude that the differences of experience-level between students have no measurable effect on the effectiveness of Eliph.

Limitation

Analysis shows that code history helps students understanding the code (Q1) and the intention of the author behind the code (Q3). However, the feedback evaluation result does not agree that the feedback from the Eliph group had better understood of their code (R1). To investigate this seemingly conflicting results, we looked at the comments within feedback students left in the assessment (Step 1). What we have found is that most of the feedback comments do not contain any visible signs from which to infer how much one did understand the author's intention. For instance, suggesting a better algorithm took the highest proportion of the comments for the *Efficiency* question:

"Your solution has $O(n^2)$ complexity as seen from the lines 43 and 44 (nested loops), while a solution having $O(n * \log(n))$ is available."

Most of the comments from *Readability* criterion were related to variable naming:

"Your variables all have proper naming clearly representing what they work for – really good."

There were only few comments that refers the author's intention or thought process:

"Thought process behind the program is clear."

Thus, as of our evaluation setting, given the set of comments, authors cannot distinguish if students understood the underlying thought process.

There are other factors related to our experiment setting that may have degraded the significance of browsing the code history. One is the fact that most of the participated students were very familiar with the programming exercises we used in our experiment. As shown in Table 1, over 80 percent of the students got 100 score by automated grading in three exercises. In addition, since the experiment was held in an introductory CS course, the exercises required relatively simple solutions, most submissions not exceeding 300 lines of code. Thus, if a further experiment is done with more complex exercises, we may be able to find more rich evidence of the effectiveness of browsing code history, yielding more definitive results.

CONCLUSION

In this paper, we introduced Eliph, a web-based peer assessment system with code history visualization, specifically designed to promote higher quality of feedback for the programming education environment. From the experiment conducted in an undergraduate CS course, we showed that looking at the character-level history visualization has multiple benefits in peer code assessment: (i) Looking at the code history helped the evaluator understand the code structure as well as the author's intention more clearly; (ii) The overall quality of feedback is higher when the code was evaluated with the code history; (iii) Evaluators felt it is helpful to look at the code history for their own learning.

During the experiment, we observed that code authors were unable to estimate if peers correctly understood the code or the underlying intention. This is because most students did not explicitly state their level of understanding in the feedback. We plan to improve the design of the assessment procedure in Eliph to make the feedback more expressive. For example, our current tool allows students to make comments only on the final revision. This can be improved by letting students to comment within the timeline of code history. In this way, students can comment on an intermediate revision with the context of the author's problem solving process.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant (#01160227).

REFERENCES

1. Nancy Falchikov. 2005. *Improving Assessment through Student Involvement: Practical Solutions for Aiding Learning in Higher and Further Education*. RoutledgeFalmer, Chapter 5.
2. Stephen Fallows and Balasubramanyan Chandramohan. 2001. Multiple approaches to assessment: reflections on use of tutor, peer and self-assessment. In *Teaching in Higher Education*, Vol. 6. Taylor & Francis, 229–246.
3. Lile Hattori and Michele Lanza. 2010. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 235–238.
4. Wu-Yuin Hwang, Chin-Yu Wang, Gwo-Jen Hwang, Yueh-Min Huang, and Susan Huang. 2008. A web-based programming learning environment to support cognitive development. In *Interacting with Computers*, Vol. 20. 524–534.
5. Suin Kim, Jae Won Kim, Jungkook Park, and Alice Oh. 2016. Elice: An online CS Education Platform to Understand How Students Learn Programming. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. ACM, 225–228.
6. Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R Klemmer. 2015. Peer and self assessment in massive online classes. In *Design Thinking Research*. Springer, 131–168.
7. Thomas D LaToza and Brad A Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*. ACM, 8.
8. Mark Liffiton. 2014. CS 127 - Fall 2014: Grading Rubric. `https://sun.iwu.edu/~mliffito/class/2014f/cs127/`. (2014).
9. Raymond Lister and John Leaney. 2003. First year programming: let all the flowers bloom. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*. Australian Computer Society, Inc., 221–230.
10. Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E Johnson, and Danny Dig. 2012. Is it dangerous to use version control histories to study source code evolution?. In *Proceedings of the 26th European conference on Object-Oriented Programming*. Springer-Verlag, 79–103.
11. Takayuki Omori, Shinpei Hayashi, and Katsuhisa Maruyama. 2015. A Survey on Methods of Recording Fine-grained Operations on Integrated Development Environments and their Applications. *Computer Software* 32, 1 (Feb. 2015), 60–80. Translated version available at `http://www.ritsumei.ac.jp/~tomori/publication/omori-survey16.pdf`.
12. Takayuki Omori and Katsuhisa Maruyama. 2008. A change-aware development environment by recording editing operations of source code. In *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 31–34.
13. Chris Piech, Jon Huang, Zhenghao Chen, Chuong Do, Andrew Ng, and Daphne Koller. 2013. Tuned Models of Peer Assessment in MOOCs. In *Educational Data Mining 2013*.
14. Karthik Raman and Thorsten Joachims. 2014. Methods for ordinal peer grading. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1037–1046.
15. Romain Robbes and Michele Lanza. 2007. A change-based approach to software evolution. In *Electronic Notes in Theoretical Computer Science*, Vol. 166. Elsevier, 93–109.
16. Mehdi SM Sajjadi, Morteza Alamgir, and Ulrike von Luxburg. 2016. Peer Grading in a Course on Algorithms and Data Structures: Machine Learning Algorithms do not Improve over Simple Baselines. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. ACM, 369–378.
17. Jirarat Sitthiworachart and Mike Joy. 2003. Web-based peer assessment in learning computer programming. In *Advanced Learning Technologies, 2003. Proceedings. The 3rd IEEE International Conference on*. IEEE, 180–184.
18. Harald Sondergaard. 2009. Learning from and with peers: the different roles of student peer reviewing. In *ACM SIGCSE Bulletin*, Vol. 41. ACM, 31–35.
19. Chengzheng Sun and Clarence Ellis. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. ACM, 59–68.
20. Keith Topping. 1998. Peer assessment between students in colleges and universities. In *Review of educational Research*, Vol. 68. Sage Publications, 249–276.
21. Yanqing Wang, Hang Li, Yuqiang Feng, Yu Jiang, and Ying Liu. 2012. Assessment of programming language learning based on peer code review model: Implementation and experience report. In *Computers & Education*, Vol. 59. Elsevier, 412–422.
22. Allan Wigfield and Jacquelynne S Eccles. 2000. Expectancy–value theory of achievement motivation. In *Contemporary educational psychology*, Vol. 25. Elsevier, 68–81.
23. YoungSeok Yoon, Brad A Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 119–126.