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

Jungkook Park
School of Computing
KAIST, Republic of Korea
pjknkda@kaist.ac.kr

Suin Kim
School of Computing
KAIST, Republic of Korea
suin.kim@kaist.ac.kr

Yeong Hoon Park
School of Computing
KAIST, Republic of Korea
yhpark92@kaist.ac.kr

Alice Oh
School of Computing
KAIST, Republic of Korea
alice.oh@kaist.edu

CSCW 2017, February 25–March 1, 2017, Portland, OR, USA.
ACM ISBN 978-1-4503-4688-7/17/02.
http://dx.doi.org/10.1145/3022198.3023266

Abstract

Peer assessment is an effective pedagogical tool in which students engage in the process of evaluating other student's work. In programming education, peer assessment involves peer code review, where students mark and give feedback other peer's code. We introduce **Eliph**, a web-based peer assessment tool 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. In a controlled experiment performed in an undergraduate CS course, we found that Eliph helps students understand code structure and the author's intention more clearly, and promotes higher quality of peer feedback.

Author Keywords

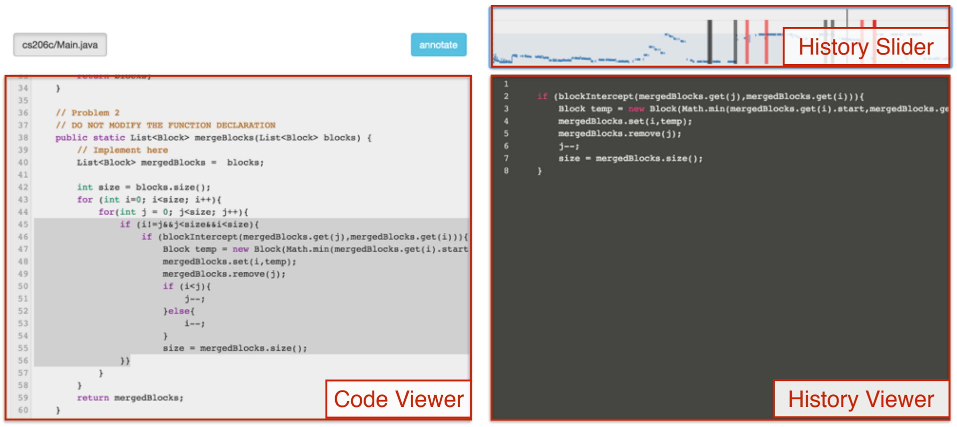
Peer assessment; Peer review; Data visualization; Time series visualization; Code history;

ACM Classification Keywords

H.5.m [Information interfaces and presentation (e.g., HCI)]: Miscellaneous

Introduction

In computer science (CS) education, peer assessment involves students reviewing other students' code and giving marks and feedback, and several studies in CS edu-



```
    }

    // 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 of the selected code block. User can drag the history slider horizontally to browse the code history. The history viewer (right) shows the code corresponding to the version indicated by the history slider.

cation found positive effects of peer assessment. Studies found that peer assessment helps students improve their programming skills [8, 5] and delivers relevant and useful feedback to students [5]. 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 [6, 1].

Eliph, a web-based peer assessment tool for programming education, visualizes the code history at a fine-grain level to assist students in peer assessment by giving 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 [2], and one of the common practices to overcome this is to use a system for browsing the history of the code.

Eliph

In this section, we describe three elements of our code peer assessment tool Eliph.

Character-Level Code History

Eliph provides visualization of code history at a fine-grain, as small as one character insertion and deletion. This is in agreement with recent research that it is import to look at code evolution at much finer grain than traditional version control systems (VCS) [3, 4]. Operation transformation (OT) is one effective approach for working with real-time document changes at a fine grain [7], and Eliph uses OT to efficiently collect, store, and re-create code revision history.

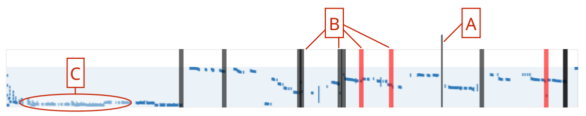


Figure 2: History slider in Eliph. The slider visualizes the timeline of code history. A: The current position of the slider, draggable with pointing device. B: Execution events; red and black color represents execution events containing error message and non-error output, respectively. C: Vertical position of the blue dots represent the location of the code change within the file.

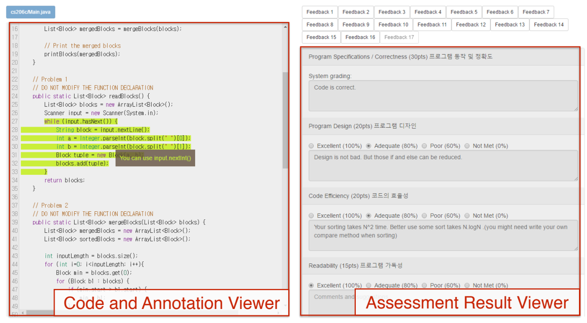
Selection-Based History Tracking
For some programming exercises, the history of the entire code can be too vast and complex to navigate, and as such, Eliph provides a way of filtering the code history based on user's selection. Users can select arbitrary block of the code, then Eliph extracts the OTs associated to the selected block and enables the users to see only relevant code history to the selected code block.

Execution Events
We define execution events as the set of all events generated from the cycle of problem solving, which consists of a series of actions such as code writing, submission, compilation, execution and grading that students repeat to solve programming exercise. Eliph visualizes the executions events to let the users refer them as clues to find out what the code author was trying to achieve at the moment.

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.

Figure 2 shows the design of the *History Slider* in detail. As student drags mouse over the slider, the corresponding version of the code is displayed in *History Viewer*. Execution events are visualized by the vertical bars, which students can click to see more detailed information about the execution event. The blue shapes visualizes where the code changes have occurred over time.



```
        List<Block> mergedBlocks = mergeBlocks(blocks);

        // Print the merged blocks
        printBlocks(mergedBlocks);
    }

    // Problem 1
    // DO NOT MODIFY THE FUNCTION DECLARATION
    public static List<Block> readBlocks() {
        List<Block> blocks = new ArrayList<Block>();
        Scanner input = new Scanner(System.in);
        while (input.hasNext()) {
            String block = input.nextLine();
            int a = Integer.parseInt(block.split(" ")[0]);
            int b = Integer.parseInt(block.split(" ")[1]);
            Block tuple = new Block(a, b);
            blocks.add(tuple);
        }
        return blocks;
    }

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

        int inputLength = blocks.size();
        for (int i=0; i<inputLength; i++){
            Block min = blocks.get(0);
            for (Block b1 : blocks) {
                if (min.start > b1.start) {
                    min = b1;
                }
```

You can use input.nextInt()

Program Specifications / Correctness (30pts) 프로그램 동작 및 정확도

System grading:
Code is correct.

Program Design (20pts) 프로그램 디자인

Excellent (100%) [✓ Adequate (80%)] Poor (60%) Not Met (0%)
Design is not bad. But those if and else can be reduced.

Code Efficiency (20pts) 코드의 효율성

Excellent (100%) [✓ Adequate (80%)] Poor (60%) Not Met (0%)
Your sorting takes N^2 time. Better use some sort takes N.logN .(you might need write your own compare method when sorting)

Readability (15pts) 프로그램 가독성

[✓ Excellent (100%)] Adequate (80%) Poor (60%) Not Met (0%)
Comments and suggestions

Figure 3: Page showing received feedback. The code and annotation viewer (left) shows the code and the annotations the evaluators created. The assessment result viewer (right) shows the markings and comments he/she received.

The received feedback is viewed in a page shown in Figure 3. Code and annotation viewer shows the code and the code annotations created by the peer evaluators. The annotations are colored as bright-green color over the annotated area in the code. Moving mouse pointer over the annotated area shows a pop-up overlay containing message of the annotation. Assessment result viewer shows markings and comments he/she received on each criterion.

Conclusion

We introduced Eliph, a peer assessment system with code history visualization for programming education. From a randomized controlled experiment we conducted in an undergraduate CS course, we found that looking at the code history helps evaluators understand author's intention more clearly and promotes higher quality of feedback. We hope to give hands-on experience to CSCW attendees who are interested not only in our research findings, but also the visualization and overall user experience of Eliph.

Acknowledgements

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

REFERENCES

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. Harald Sondergaard. 2009. Learning from and with peers: the different roles of student peer reviewing. In *ACM SIGCSE Bulletin*, Vol. 41. ACM, 31–35.
7. 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.
8. 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.