Measuring an Impact of Block-Based Language in Introductory Programming

Yoshiaki Matsuzawa[1](✉), Yoshiki Tanaka[2], and Sanshiro Sakai[2]

[1] Aoyama Gakuin Univeristy, 5-10-1 Fuchinobe, Sagamiahra, Kanagawa, Japan
matsuzawa@si.aoyama.ac.jp
[2] Shizuoka University, 3-5-1 Johoku, Hamamatsu, Shizuoka, Japan

Abstract. The use of block-based visual language in introductory programming is a popular method in education. However, there is little research which provides evidence showing advantages of block-based language. This paper presents the results of learning data analysis with fine grain logs recorded by students' development environment where the students can select their language in block-based or Java. A total of 400+ students' logs collected each of four years were analyzed. The results show that migration from Block to Java can be consistently seen each year, although the whole block-editing rate was influenced by the method of the instructor's introduction. Though block-editing did not affect working time and Lines of Code (LOC), it could reduce the compile error correction time, whereas using Java requires approximately 20% of compile error correction time for students. We concluded that block-based language worked to encourage students to focus high-level algorithm creation, as well as it provides an advantage to understanding text-based language.

Keywords: Programming education · Block-based language · Learning analytics · Working time analysis · Compile error analysis

1 Introduction

The body of introductory programming is not to develop an understanding of the grammar of particular programming languages, but it should develop the problem-solving skills with computing that is called "computational thinking" [1]. The similar concept has been proposed by the United Nations Educational, Scientific and Cultural Organization (UNESCO), as "designing a task-oriented algorithm" [2]. Both statements include common sense. One is to focus on the thinking and creation of algorithms. Another is that computing is not only dependent upon the use of actual computers, but logical modeling for the required problem-solving.

A use of visual language, especially "building-block approach" [3] is the most popular way to form the learning environment for the purpose of education. Many block-based languages for education have been proposed over more than

T. Brinda et al. (Eds.): SaITE 2016, IFIP AICT 493, pp. 16–25, 2016.
DOI: 10.1007/978-3-319-54687-2_2

two decades, and yet developers continue by trial and error to improve the language using modern software technology. The first workshop specifically block-based language focused was held in the last year (Blocks and Beyond in Visual Languages and Human-Centric Computing (VL/HCC 2015) [4]). The workshop was able to collect a remarkable number of submissions, the participants discussed the design of the next generation of block-based language, including the topic of how to design the tools as a bridge to text-based language (e.g. [5,6]).

However, as the workshop stated "Despite their popularity, there has been remarkably little research on the usability, effectiveness, or generalizability of affordances from these environments" [4], there is little research which provides evidence showing advantages of block-based language. Practitioners using visual programming languages believe that the visual programming approach is an effective way in developing computational thinking because learners can focus on their problem-solving tasks [7]. The block-based language should be used as scaffolding to text-based language. However, the belief is not verified, and the percentage of degree is not clear yet.

Hence, we tried learning data analysis with fine grain logs recorded by students' development environment. We conducted the introductory education using the tools we developed, where the learners can switch their language between visual-block language (Block) and one for a text-based language (Java) by bidirectional translation technology. The data analysis was conducted mainly using the data of time students spent. These include amount of total working time, the time using block-editor, the time of compile error correction. The tile representation of each student shows gradual migration to Java on their own schedule, and reducing compile error correction time shows the success by focusing on high-level algorithm creation.

2 Related Work

There have been many reports of using block languages for introductory programming education [8,9], as well as reports on the development of new block languages [10–14]. Researchers agree that block-based languages feel familiar to beginners, however they are not a standard for use in introductory programming. Not a few researchers/practitioners claim disadvantages of block-based language, particularly that block-based language is different from common practical languages used in industry, so that students have difficulty in moving to text-based programming after their introductory programming course.

Hence, which text-based or block-based language we should use in introductory programming is still an unsolved theme in this field. Lewis et al., conducted direct comparison using Scratch and Logo [15], however, their results were very limited to illustrate the advantage of block-based language. Matsuzawa et al. demonstrated gradual migration from block-based to text-based language using a bidirectional translation system [16]. Although the limitation of the study is that they showed only trajectory of their working time, they demonstrated the capability of transition to text-based language from block-based language.

From the viewpoint of tool proposition, a rich number of tools have been proposed on migration from block-based languages or text-based languages, and it is still growing. Pasternak [17] proposed offering the ability to translate a program written in Block to a program written in Java. Google Blockly [14] can be translated into multiple languages (e.g., into Java-script or Python) from its block representation. In 2012, Dann et al. tackled the problem of figuring out how students can transfer their knowledge of Alice3 (a block-based language) to Java programming, and they reported that experience with a block language behaved as a helpful scaffolding for students learning to program in Java [18].

The present research projects are attempts to raise the "ceiling of programming" up to practical programming levels. One approach is to raise the ceiling of block-based language. Harvey et al. [19] proposed a functionality they called BYOB ("Build Your Own Block") in Scratch. This functionality expands the descriptive capabilities of the block language. The purpose of the project is to support a block language suitable for a wide range of users, from beginners to professionals. The developers of Scratch and Squeak are currently working on a GP project [6]. GP will be the first block-based language written by itself, where learners can explore the entire software system by the level of learners' interest. It means the tool will provide an open-ended, no ceiling environment, where either text-based or block-based is merely one of the representations.

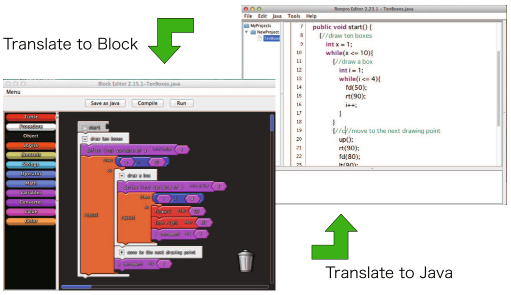
Another approach is to promote migration from block-based to text-based language by making educational scaffolding on the tools. The most popular approach is considered to develop language translation systems. Warth et al. proposed a bidirectional translation system, "TileScript", between a block language and JavaScript [20], although that research verified only that the technical requirements could be met to implement the prototype system. Matsuzawa et al. demonstrated a mutual translation between Block and Java [16]. Similar attempts are growing a variety of languages. BlockPy [5] is designed for education in Python. PencilCode [21] is specially designed for migration to text-based language using modern Javascript technology.

Improvement of the editing environment of text-based language is another approach. One distinction of the BBC micro:bit project described above is providing an intermediate layer of editing system called TouchDevelop, which provides some scaffolding for text editing and, is designed for the migration to text-based language. Homer et al. proposes Tiled Grace, which is a tiled representation of Grace and, was an originally designed text-based language for education [22]. Kolling et al., who is the founder of GreenFoot, proposes the "Frame-Based Editing" [23] where the "frame" is added to support novices with Java.

3 Method

3.1 Tool: Bidirectional Translation System

We conducted empirical studies in our introductory programming class over the span of four years. The goal of the study was to evaluate the impact of



```
public void start() {
  {//draw ten boxes
    int x = 1;
    while(x <= 10){
      {//draw a box
        int i = 1;
        while(i <= 4){
          fd(50);
          rt(90);
          i++;
        }
      }
      {//c//move to the next drawing point
        up();
        rt(90);
        fd(80);
        lt(90);
```

Fig. 1. Bidirectional translation system between Block and Java.

introducing block-based language. In each year, we used a bidirectional translation system between Block and Java proposed in the previous study [16]. The resulting environment has two interfaces, one for the (visual) Block language and one for the textual Java language, as well as a system for bidirectional translation between the two languages. Figure 1 shows the user interface for the environment. All the students in our class were given the opportunity to select the language they used to solve their programming assignments.

3.2 Research Question

In the previous study, Matsuzawa et al. demonstrated that learners would choose to use Block first, which will act as scaffolding for learning programming, and then learners will gradually migrate to Java on their own schedule. We expanded the research to explore the further impact of the environment. The original two research questions for this paper are as follows:

RQ1 Can the gradual migration nature be seen consistently every year, even if some variables change (students, teachers, method of teaching, or tool functionality)?

RQ2 Does the tool (BlockEditor) successfully encourage students to focus their algorithm creation by removing compile error correction opportunity? If so, how much wasted time can we save?

3.3 Educational Environment Descriptions

The introductory programming course was designed for art students, rather than for computer science students. Therefore, the objective of the course was to develop an understanding of task-oriented programming. The objective was independent from any programming language, although Java language was used in the actual environment. Approximately 100 students participated in this course each year; two lecturers and six teaching assistants conducted the class.

Because of RQ1, we conducted an action research. Actions taken each year in our experimental course were listed as follows:

1st year (2012) The first year we started to use BlockEditor with the Block-Java translation system.

2nd year (2013) Major improvements in BlockEditor were made, especially the functionality of the variable scope indicator and the design of the block for method creation. The improvement was made using the method calling navigation system MeRV [24].

3rd year (2014) Main lecturer changed from the author of Block Editor to a teacher who was from outside the research. The lecturer used Java using sample codes, whereas the former lecturer used Block for explanation.

4th year (2015) Compulsory assignments using Block was reduced from 4 times to once. The opportunity for everyone to use Block was reduced.

3.4 Metrics

All the students' activities in the development environment were recorded by PPV (Programming Process Visualizer) [25], and the logs used to calculate metrics to analyze the learning process. We calculated five kinds of metrics as follows for both RQ1 and RQ2.

Working Time (WoT) The working time for each assignment. It was calculated by summing up the time and excluding periods of longer than 5 min with no user operation in the development environment.

Block Editing Working Time Ratio (BWT%) The ratio of working time to block out of the total working time. We can assume students used Java outside of BWT.

$$BWT\% = \frac{BWT(WorkingTimewithBlock)}{WoT} \tag{1}$$

Compile Error Correction Time (ECT) Compile Error Correction time calculated by activity logs for each compilation error occurrence. Further description will be provided below.

Compile Error Correction Time Ratio (ECT%) The ratio of compile error correction time out of the total working time.

$$ECT\% = \frac{ECT}{WoT} \tag{2}$$

Lines of Code (LOC) Lines of Code for finally submitted assignment.

The technical description of the calculation method for ECT was provided by this paper [26]. The difference in time between the error occurred and resolved, will be calculated in a general case. However, sometimes multiple errors occur or are resolved at the same time. For such cases, the system calculates by the amount of time spent in error collection by the number of errors. An assumption of the method is the difficulty of corrections is equal in every case.

Table 1. Descriptive statistics for the calculated metrics.

Year	#student	#task	BWT%	WoT	ECT	ECT%	LOC
2012	102	36	36.7(17.5)	47.9(15.8)	6.8(2.9)	14.3(4.7)	57.9(14.4)
2013	96	38	45.9(23.3)	51.7(25.3)	5.3(2.6)	12.2(4.6)	66.3(36.6)
2014	106	34	29.6(21.8)	44.6(14.4)	5.9(3.5)	14.6(5.9)	65.5(59.1)
2015	100	48	19.2(23.7)	43.4(15.5)	5.8(3.0)	15.1(5.5)	65.7(49.2)
Total	404	156	32.7(23.7)	46.8(18.4)	6.0(3.1)	14.1(5.3)	63.8(43.4)

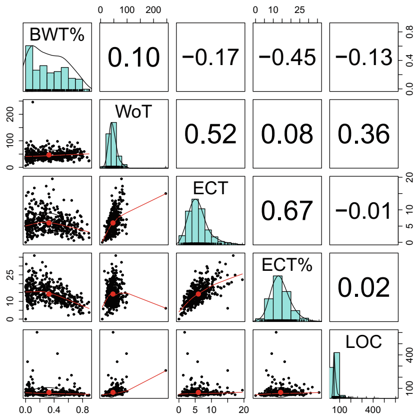


Fig. 2. Correlation table between 5 metrics

4 Results

4.1 Results of Statistics

Descriptive statistics for all calculated metrics are summarized in Table 1. For each metric, the number shows the average of the value, and the number in parenthesis shows the standard deviation of the distribution. The unit of value for BWT% and ECT% is the percentage amount per hundred, for WoT and ECT is minutes, for LOC is the number of lines, respectively. The average of BWT% was 32.7 for all year, although the data for each year indicates the variations in value. The total average of ECT% was 14.1. If we assume that block can swipe ECT, and the fact of BWT%, we estimate the ECT% using Java as approximately 20%.

We calculated Pearson Correlation for all combinations of the 5 metrics, the results are shown in Fig. 2. The most significant result is BET% has a negative correlation with ECT%, which clearly indicates the Block-editing was able to reduce the compile error correction time. The results also indicate Block-editing did not affect working time and LOC, which means that the quality of the outcome by Block-editing is equivalent to the outcome by text-editing.

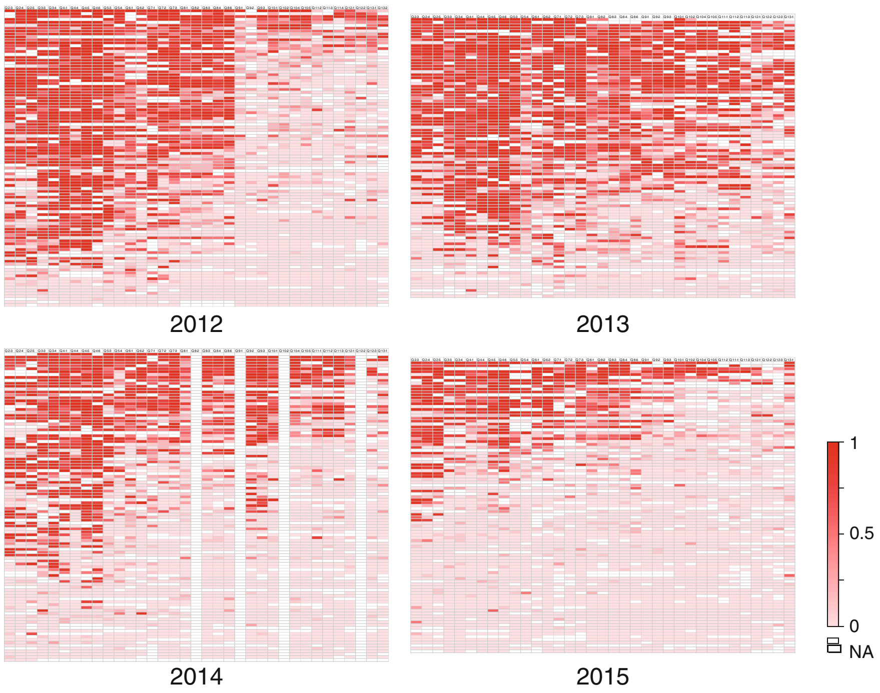


Fig. 3. Grid representation of BWT% (Block Editing Rate) for each student.

4.2 Grid Representations

We created a grid representation of the rate of BlockEditor using (BET%) to illustrate the nature of the seamless migration from Block to Java. The representation is shown in Fig. 3. Each column represents one task assigned to students, arranged in chronological order from left to right. Each row represents one student; the rows are sorted by course-average BET% with higher rows indicating higher average use. Each cell represents the value of BET% for a particular student completing a particular task. High-intensity color indicates a high BET%, and low-intensity indicates an BET% of 0 (i.e., using only Java). We selected 36 tasks which were mostly common for all four years, however three of the tasks selected were lacking in 2014. The tasks can be seen as vertical white belts in 2014.

We can observe that the seamless migration both overall (by the number of students who used Java in the class) and at the individual level for all 4 years were consistent. A sudden drop of the whole rate could be seen in the middle of 2012; however, it improved to a milder migration in 2013. As the class content involved introducing method/function, the improvement was caused by the action of improvement of tool functionality, as we described in Sect. 3.3. Although the pattern of gradation of migration can be seen each year, the entire density is different; 2014 and 2015 are low-density. The density corresponds to the average ratio of the year as shown in Table 1. In the classes of 2014 and 2015, we could

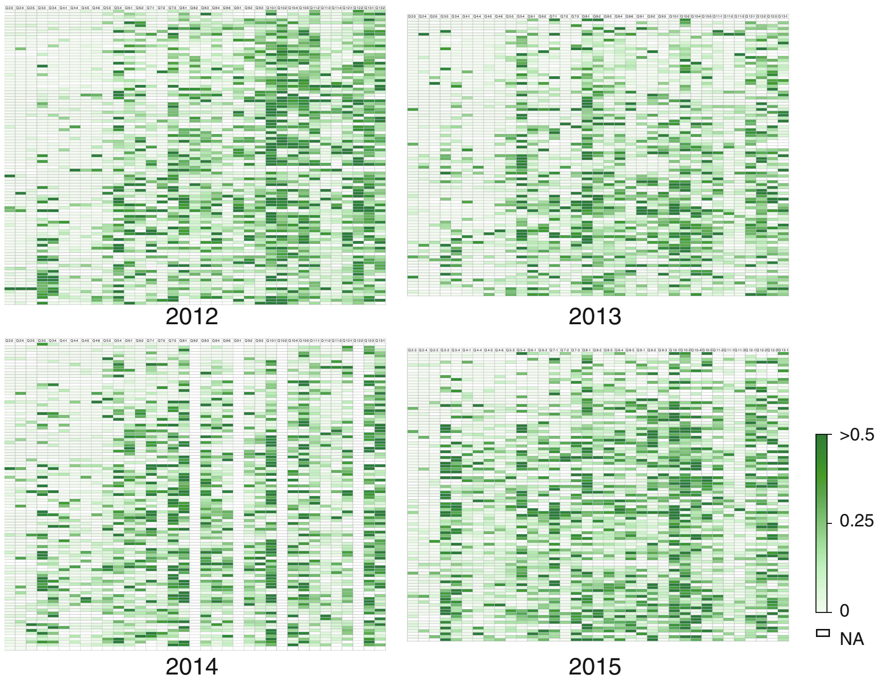


Fig. 4. Grid representation of ECT% (Compile Error Correction Rate) for each student.

not observe any other difference with the exception of the actions described in Sect. 3.3; the cause of the difference is influenced by the method of the instruction by the teacher, and opportunities to use Block.

We created another grid representation using Compile Error Correction Time Rate (ECT%). The representation is shown in Fig. 4. When we compare Figs. 3 and 4, we can observe that inverted illustration. This means the Block reduces the compile error correction rate, the result corresponds with the result of statistic correlation (Fig. 2) and our intuitive sense. Cells with high ratio focus on the latter part of the class (right-side on charts), because more advanced grammar content (e.g. Method/Function, collection) was introduced. Another significant result we can observe is that the factor of task was bigger than the BET% in the past. For example, the latter part of 2012 clearly illustrates that density of ECT% shows no difference between the high BET% group and the low BET% group.

5 Discussion

RQ1 asked "Can the gradual migration nature be seen consistently each year, even with some variable changes?". The results were generally positive: we can consistently see the migration nature. These results reinforced evidence that a block language can successfully act as scaffolding for students learning text-based programming. However, the factor of the method of instruction, especially which language is used in the explanation in the lecture affects the selection of language. The impact can be calculated by comparing the lowest 19% of BET% (2015) to the highest 46% (2013). The result indicate that not only the tools, but also the educational design including the use of tools is important.

RQ2 asked "Does Block encourage students to focus their algorithm creation by removing compile error correction opportunity?". Both statistics and micro-analysis using grid representations indicate that Block clearly eliminates compile error correction time. The result itself is not surprising; however, it is not a small thing that the research revealed the amount (approx. 20%) of overhead which was wasted for compile error correction. We can eliminate 20% of overhead by Block, also can reduce to 10% in mixture usage. Additional observation shows that some students select Block-editing after being plagued with compile error correction. Hence, we concluded that block-based language worked to encourage students to focus on high-level algorithm creation.

The lack of direct evidence of the learners' understanding is still a limitation and requires further consideration. We believe that the results could reinforce existing evidence to encourage teachers who use a block language in introductory programming education.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Numbers JP25730203, JP26280129.

References

1. Wing, J.: Computational thinking. Commun. ACM **49**(3), 33–35 (2006)
2. UNESCO: ICT Curriculum for School/Program of Teacher Development (2002). http://unesdoc.unesco.org/images/0012/001295/129538e.pdf
3. Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., Resnick, M.: Scratch: a sneak preview. In: Proceedings Second International Conference on Creating Connecting and Collaborating Through Computing, pp. 104–109 (2004)
4. Turbak, F. (Chair): Blocks and beyond: lessons and directions for first programming environments. http://cs.wellesley.edu/~blocks-and-beyond/. Accessed 3 Feb 2016
5. Bart, A., Tilevich, E., Shaffer, C., Kafura, D.: Position paper: from interest to usefulness with blockpy, a block-based, educational environment. In: Blocks and Beyond Workshop (Blocks and Beyond), pp. 87–89 (2015)
6. Monig, J., Ohshima, Y., Maloney, J.: Blocks at your fingertips: blurring the line between blocks and text in GP. In: Blocks and Beyond Workshop (Blocks and Beyond), pp. 51–53 (2015)
7. Fal, M., Cagiltay, N.: How scratch programming may enrich engineering education. In: 2nd International Engineering Education Conference (IEEC 2012), pp. 107–113 (2012)
8. Lewis, C.: What do students learn about programming from game, music video, and storytelling projects? In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE 2012), pp. 643–648 (2012)
9. Ozoran, D., Cagiltay, N., Topalli, D.: Using scratch in introduction to programming course for engineering students. In: 2nd International Engineering Education Conference (IEEC 2012), pp. 125–132 (2012)

10. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of squeak, a practical smalltalk writtern in itself. In: Proceedings of ACM OOPSLA 1997, p. 318 (1997)
11. Scratch Team Lifelong Kindergarten Group MIT Media Lab: Scratch - imagine.program.share-. http://scratch.mit.edu/
12. Cooper, S., Dann, W., Pausch, R.: Teaching objects-first in introductory computer science. In: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2003, pp. 191–195. ACM, New York (2003)
13. Cheung, J.C., Ngai, G., Chan, S.C., Lau, W.W.: Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students. In: SIGCSE 2009 Proceedings of the 40th ACM Technical Symposium on Computer Science Education, New York, NY, USA (2009)
14. Google Inc.: Blockly: a visual programming editor. http://code.google.com/p/blockly/. Accessed 17 Mar 2013
15. Lewis, C.: How programming environment shapes perception, learning and goals: logo vs. scratch. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE 2010), pp. 346–350 (2010)
16. Matsuzawa, Y., Ohata, T., Sugiura, M., Sakai, S.: Language migration in non-cs introductory programming through mutual language translation environment. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE 2015, pp. 185–190 (2015)
17. Pasternak, E.: Visual programming pedagogies and integrating current visual programming language features. Master's thesis, Carnegie Mellon University Robotics Institute Master's Degree (2009)
18. Dann, W., Cosgrove, D., Slater, D., Culyba, D., Cooper, S.: Mediated transfer: Alice 3 to java. In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, SIGCSE 2012, pp. 141–146 (2012)
19. Harvey, B., Monig, J.: Bringing no ceiling to scratch: can one language serve kids and computer scientists? In: Constructionism 2010, Paris (2010)
20. Warth, A., Yamamiya, T., Ohshima, Y., Scott, W.: Toward a more scalable end-user scripting language. In: Proceedings Second International Conference on Creating Connecting and Collaborating Through Computing, pp. 172–178 (2008)
21. Bau, D., Bau, D.A., Dawson, M., Pickens, C.S.: Pencil code: block code for a text world. In: Proceedings of the 14th International Conference on Interaction Design and Children, pp. 445–448 (2015)
22. Homer, M., Noble, J.: Combining tiled and textual views of code. In: 2014 Second IEEE Working Conference on Software Visualization (VISSOFT), pp. 1–10 (2014)
23. Kölling, M., Brown, N.C.C., Altadmri, A.: Frame-based editing: easing the transition from blocks to text-based programming. In: Proceedings of the Workshop in Primary and Secondary Computing Education, WiPSCE 2015, pp. 29–38 (2015)
24. Ohata, T., Matsuzawa, Y., Sakai, S.: Merv: a scaffold to promote creating 2D map of method call structure in block-based programming language. In: IFIP TC3 2015, pp. 352–362 (2015)
25. Matsuzawa, Y., Okada, K., Sakai., S.: Programming process visualizer: a proposal of the tool for students to observe their programming process. In: Innovation and Technology in Computer Science Education (ITiCSE 2013), pp. 46–51 (2013)
26. Hirao, M., Matsuzawa, Y., Sakai, S.: Compile error collection viewer: visualization of learning curve for compile error correction. In: IFIP TC3 2015, pp. 310–309 (2015)