Why Novice Programmers Fall into a Pitfall?: Coding Pattern Analysis in Programming Exercise

Kenji Fujiwara*, Kyohei Fushida†, Haruaki Tamada‡, Hiroshi Igaki§ and Norihiro Yoshida*
*Graduate School of Information Science, Nara Institute of Science and Technology
Takayama-cho 8916–5, Ikoma-shi, Nara, 630–0192 Japan
{kenji-f,yoshida}@is.naist.jp
†Research and Development Headquarters, NTT DATA Corporation
Toyosu Center Bldg. Annex, 3-9, Toyosu 3-chome, Koto-ku, Tokyo, 135–8671 Japan
fushidak@nttdata.co.jp
†Faculty of Computer Science and Engineering, Kyoto Sangyo University
Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, 603–8555 Japan
tamada@cse.kyoto-su.ac.jp
‡Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-shi, Osaka, 568–0871 Japan
igaki@ist.osaka-u.ac.jp

***Abstract*—It is crucial for educators to understand pitfalls for novice programmers in programming education at computer science course. By giving well-timed advice to students who fall into a pitfall, educators are able to achieve efficient teaching and keep up their students' motivation. However, so far, it is less well-understood how novice students edit source code and why they fall into a pitfall. In this paper, we analyzed coding patterns of novice students empirically. We collected programming activities by students on exercises of programming course, and then performed qualitative and quantitative analysis . In qualitative analysis, experienced programmers analyzed patterns of the novice programmers manually. In quantitative analysis, we focused on transtions of the edit distance between a source code of each student under development and a correct source code in a programming class. As a result, we confirmed coding patterns of novice programmers in case of falling into pitfalls, and the characteristics of transitions of edit distance metric in the case that novice students had faced difficulty in understanding the exercise.**

***Keywords*-Programming education; metrics; edit distance**

I. Introduction

Software is widely used and become increasingly influential in the world. In order to develop high quality software, it is necessary to train superior software developers with adequate programming skills. For this reason, most computer science course of universities and colleges teach programming.

On programming exercises in a class, novice students often encounter problems due to lack of their understanding about programming syntax or concept[1]. We consider detailed causes of these problems come from lack of understanding about exercises and programming languages. When students are stuck in a pitfall, giving well-timed advice to them allow efficient learning and keep up their motivation. In general, two or three teachers instruct programming classes to thirty or forty students. Therefore, it is difficult to capture programming activities of all students and help them when needed.

It is beneficial for educators who design and teach programming exercises to have knowledge about pitfalls that novices typically experienced. Educators who have the knowledge are able to give novices well-timed good advices. Also, once novices get lectured on typical pitfalls in advance, they can easily avoid the pitfalls.

Our research goal is aimed to clarify coding patterns that novices typically experiences in case of falling into pitfalls. There are typically two approaches to observing how students fall into a pitfall. The first one is analyzing manually how students complete a programming exercise. This approach is promising to analyze correctly the reason why modifications and errors occurred in source code. The other one is analyzing modifications to source code (e.g., analyzing transitions of source code metrics). This approach can be automatically done without manual inspection.

Several studies investigated processes of the program comprehension of the novice programmers[2]. Those studies also investigated processes of the novice programmers and reported differences between novices and experts[3], [4]. To support programming education, we analyzed modifications of the source code written by novices.

As an exploratory study, this paper presents quantitative and qualitative analyses of code modification done by novice programmers. In our analyses, we collected a version of source code every compilation and saving. For the qualitative analysis, the experienced programmers (the 1st and 2nd authors) determined whether or not each modification to source code is appropriate by manual inspection for each of the

versions and the compile error messages. For the quantitative analysis, we analyzed evolutions of edit distances of token sequences between developing and correct source code.

II. Methodology

In this paper, we define how novices process programming exercises as a coding pattern. It is not clear how novices complete a programming exercise. We perform qualitative and quantitative analysis to student's programming activities data collected at a programming class. In this section, we describe methodology of our analysis.

A. Qualitative Analysis by Experienced Programmers

In qualitative analysis, experienced programmers manually analyze how novices complete exercises. Figure 1 shows the concept of our qualitative analysis. Experienced programmers use source code and results of the compilation which collected when a student compile the source code. A result of the compilation contains compiler error messages and the execution code. They check from two points of view: (a) What did student modified source code from the previous version? (b) Is the modification correct or not? By using results of the checking, we confirm the type of the pitfall which student fell into.

Compiler error messages is not enough to reveal how students fall into a pitfall. Hence, we consider semantic errors of source code in qualitative analysis. We want to reveal how students faced issues in exercises and how they figure out their issues but also whether students completed exercises or not. We perform qualitative analysis by checking their edits by experienced programmers.

B. Quantitative Analysis

In quantitative analysis, we measure and analyze product metrics from source code which is created in the process of solving the exercises. Targets of the measurement are versions of source code which collected at every compilation same as qualitative analysis. In this paper, we introduce a **token-based edit distance** metric to measure source edits in fine-grained detail.

Figure 2 illustrates the concept of measurement of token-based edit distance. To calculate token-based edit distance, we perform lexical analysis to two source code. Generally edit distance represents a degree of difference among two string. It is defined as the minimum number of operations to edit such as additions and deletions of a character needed to transform one string into the other.

In order to calculate token-based edit distance, we tokenize source code based on lexical analysis. Tokenization in our research conforms to the following rules:

- ignore all blank characters and comments.
- encode identifiers and reserved words, literals as one character.
- not consider difference about kinds of token(such as types and literals) and scopes.

For instance, source code in Figure 3(a) will be transformed as in Figure 3(b)[1]. Thus, token-based edit distance is calculated by measuring edit distance among two versions of tokenized source code.

The aim of the analysis based on the proposed metrics is to detect novices who would fall into a pitfall without semantic analysis of source code. For example, when it takes novice programmers a long time to understand exercises, they would never edit source code. In other case, when answer for the excercise is given, token-based edit distance between answer and a version would not changed though token-based edit distace between a version and the previous one is increased. In this case, we consider he/she has made an incorrect modification. Thus, we observe transitions of metrics for source code and its edit history.

III. Analysis of Coding Pattern

We performed qualitative and quantitative analysis that followed the strategy described in section II. Our two objectives in this research are following:

- to reveal pitfalls for novices of programming
- to reveal statuses of learning, and trends of source code of novices faced to problems

[1]In Figure 3, blanks are remained for readability.

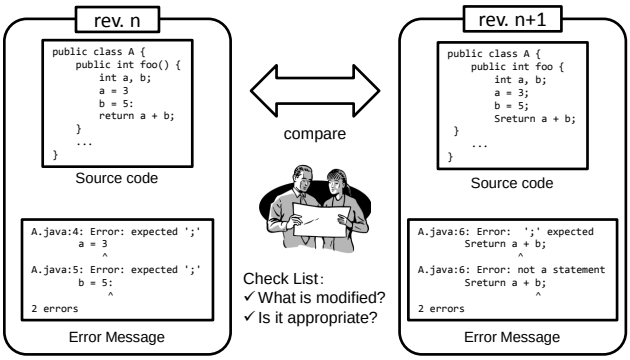

Figure 1. Concept of Qualitative Analysis

```
public class A {
    public int foo() {
        int a, b;
        a = 3;
        b = 5;
        return a + b;
    }
}
```

(a) Before tokenization

```
$A $B $C {
    $A $D $E() {
        $D $F, $G;
        $F = $H;
        $G = $I;
        $J $F+$G;
    }
}
```

(b) After tokenization

Figure 3. An example of tokenization

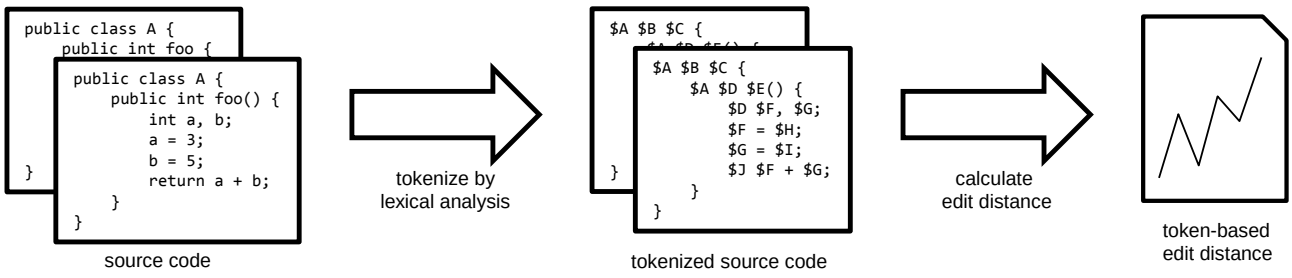

Figure 2. Calculation of Token-based Edit Distance

Table I
ANALYSIS TARGETS

data set	Language	#Students	#Problems	#Versions
data set A	C	3	10	191
data set B	Java	37	7	6428

In this section, we describe analysis targets and procedure of analysis.

A. Analysis Targets

In this paper, we analyzed two data sets. Both data set A and B target novices of programming who have no previous knowledge about the programming language. It is possible for novices to solve the problem by using fundamental functions of programming language, and exercises do not require any external library. In addition, each exercise provides a template of the source code for an answer. Novices solve the exercise by adding and modifying sentences to the template source code based on specification of the exercise. Students should write source code for answer in the limited time in exercise.

Data set A was collected programming activities at a programming class that targets third year undergraduates in the department of information science. In order to collect programming activity, we used a supporting framework on programming classes implemented by third author[5]. This framework runs as a GNU C compiler (GCC) wrapper. It collects a version of source code and compiler error messages when a student compile the source code(when gcc command is executed). The programming class on data set A is not a regular class but is a part of the experiment of that framework.

Data set B was collected programming activities in a programming class for the first year undergraduates in the department of information science. Data set A and B were collected at different colleges. In data set B, in order to collect programming activity, a programming process visualization system which made by a group of fourth author was used[6]. That system was implemented as a web system and it automatically collects versions of source code when system captures programming activity by students , such as open a file, compile the source code, and execute a program. The system also collects a version of source code by the minute. Number of versions in Table I includes those automatic (unintended) savings. We use a data set collected at twice classes for analysis. In this data set, example answers and programming activities about support for students are available.

In this paper, we use data set A for qualitative analysis because the amount of data set A is appropriate for manual inspection. On the other hand, we use data set B for quantitative analysis. Data set B contains example answers and support logs as mentioned before to verify quantitative trends.

B. Procedure of Analysis

We performed analysis adopting the following procedures as mentioned in section II.

1) Qualitative analysis: First and second author manually analyzed about modification of source code and its validity. These analysts have experiences of teaching programming exercises in C language. They checked source code manually about the following information in the analysis.

- contens of the source code
- difference information based on GNU diff
- compiler error messages
- execution result (if compilation was successful)

We analyzed the following three points of view by using the above information.

Cause of the error: If the source code include errors, described the detail of error by natural language.

Contents of modification: Describe the detail of modification in natural language.

Validity of modification: Check whether modification followed to the exercise and introduced errors.

2) Quantitative analysis: We measure source code metrics and calculate some statistics of data set B. First, we calculated the following three token-based edit distance

Table II
RESULT OF QUALITATIVE ANALYSIS

Student	#Exercises	#Correct	edit		
			#Valid	#Erroneous	total
A1	10	3	20	38	92
A2	10	10	19	7	44
A3	10	10	15	13	24

metrics from each version[2]:

M_A vs. example answer
M_B vs. answer of himself/herself (if answer is correct)
M_C vs. previous version

Second, we calculated rank correlation about the three metrics for each person and exercise. Finally, we focused on characteristic exercise of the person, and then we analyzed in detail with support logs.

IV. RESULT AND DISCUSSION

A. Coding Pattern about Errors

Table II shows the result of qualitative analysis. All students completed 10 exercises. #Correct is a number of exercises that students answered correctly. #Valid is a number of editing that the analysts judged as valid. #Erroneous is a number of editing that the analysts judged as it introduces errors.

All students in data set A have fundamental knowledge and experience of Java programming. Student A2 have experiences of script language such as Ruby. We confirmed and disscuss the following patterns based on thoese information.

1) Continue to edit the part of codes which have nothing to do with the correct answer: We confirmed that students did not edit the parts of code which should be modified, but continued to edit the part of codes which have nothing to do with the correct answer. That phenomena were observed in particular student A1. We consider that experienced programmers would be able to identify the part to be fixed according to error messages. However, it is difficult for novice programmers to identify the such part because they would not have enough knowledge and experience to understand compiler error messages.

In order to support novices, it is efficient to educate preliminarily about the meaning of error messages and strategies to narrow a part to be fixed. In section IV-B, we discuss about the quantitative method to identify novices who fall into such "pattern".

2) Obsession with knowledge about another programming language: We observed that novices tend to use features of programming languages which they have already know. In fact, they use features which Java do not have but other programming language has. Likewise, they also conform to the program syntax of the other programming language which novices have already know. In most cases, that behaviors induced the errors. In particular, concatenation of string were frequently observed. Java language supports concatentation of string by plus operator. C language does not support that feature. we consider novices take a long time to understand differences between the specification between the language which is used in the exercise and is known to him/her. Thus, giving the differences about specifications would be able to realize efficient teaching.

[2]To calculate token-based edit distance metrics, we downloaded the grammar file for Java 1.5 published in Repository of JavaCC grammars[7], and modified it. Then, we implemented the parser by using JavaCC[8] based on that grammar file.

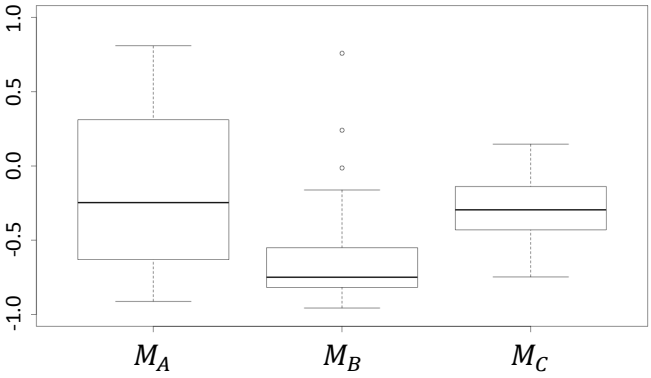

Figure 4. Distribution of correlations of the metric transitions

```
class Report13_1a2
{
  public static void main(String[] args) {
    String str = "148737438777432";
    int x = Integer.parseInt(str.substring(0,4));
    x += Integer.parseInt(str.substring(5,9));
    if(x % 3 == 0)
      System.out.println("OK");
  }
}
```

(a) An example anwser

```
class Report13_1a2
{
  public static void main(String[] args){
    String str = "148737438777432";
    String str2 = str.substring(0,4);
    String str3 = str.substring(6,10);
    int y = Integer.parseInt(str2);
    int z = Integer.parseInt(str3);
    int x = y+z;
    if(x % 3 == 0)
      System.out.println("OK");
  }
}
```

(b) An example of student's answer

Figure 5. An example of high token-based edit distance case

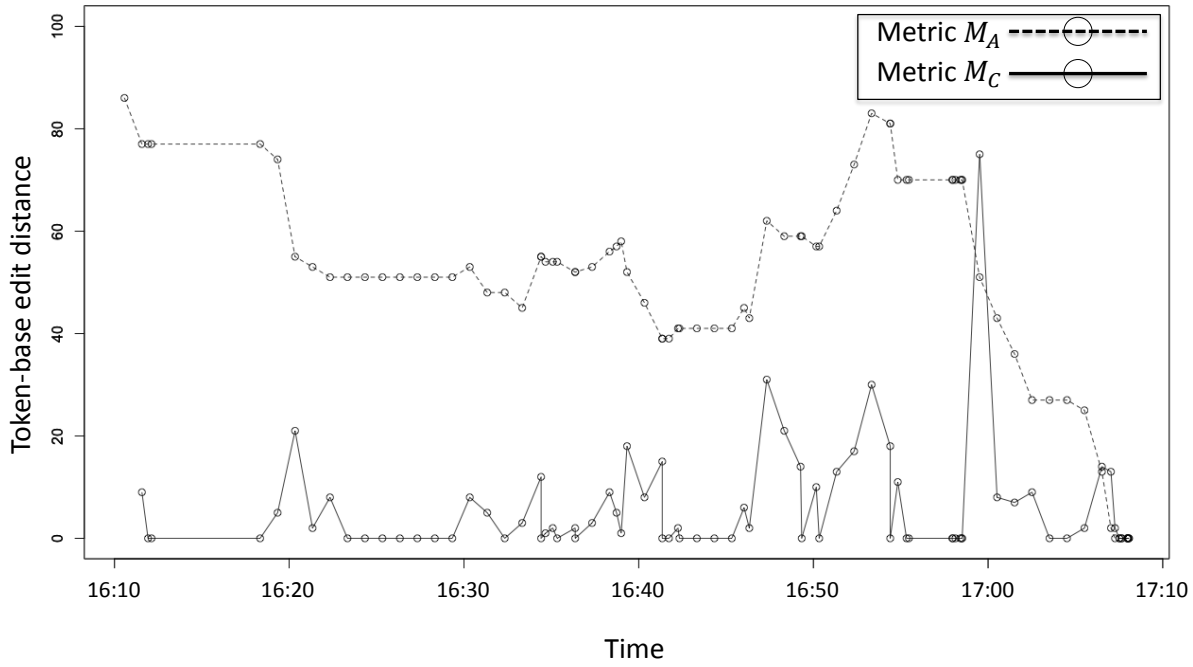

Figure 6. Transitions of edit distances about example answer and previous editing

B. Analysis of token-based edit distances

Figure 4 shows the distribution of correlations about three metrics calculated from data set B, described in III-B2. Each boxplot is a distribution of Kendall tau rank correlation coefficients between sequence of token-based edit distance metric per exercises and students, and time sequence.

About metric M_A, while many data are negatively correlated with time sequence, there are also many data which are positively correlated. In those positive correlated data, most of the cases, students answered correctly. We consider students' answer are satisfied the specification of the exercise, but these answers have different structure from the example answer, respectively. Figure 5 is an example of that case. Whited regions are provided as a template and grayed regions are example answer and students' answer.

About metric M_B, many data are negatively and strongly correlated with time sequence because the metric compares a version of the source code with the final version. Obviously, metric M_B will reach zero at the end. Nevertheless, few data are positively correlated. In these data, a student submitted his/her correct answer, and then he/she complete next exercise. Unfortunately, he/she edits previous source code as next source code.

As mentioned before, novices sometime **continue to edit the region which is unrelated to fix**. When a novice take that behavior, we expect that the metric between an example answer and the version of the source code do not decrease while the metric between a previous version and current version takes certain value. In fact, we observed the pattern which represent that behavior shown in Figure 6. Figure 6 is a transition of token-based edit distances about an exercise completed by a student. Horizontal axis represents time and vertical axis represents the value of token-based edit distance. Metric M_C is plotted as a solid line and metric M_A is plotted as dashed line. As shown in Figure 6, metric M_C sometimes takes high values during the period from 16:45 to 16:55. That means the student spend much effort to edit the source code. However, metric M_A was increased in that time. That case means the possibility of inappropriate editing by the student. From the support logs, we confirmed that a teacher supported the student because he/she had not understood the feature to solve the exercise.

We consider that teachers can support novices by observing the edit distance between example answer and them source code. Compared with line additions and deletions, token-based edit distance allow observation in more fine-grained. On the other hand, our token-based edit distances are affected by order of the methods, and difference of the identifier. For this reason, as mentioned before, token-based edit distance between the example answer and a version of the source code sometimes takes high value despite student has wrote correct source code. To improve this problem, we are considering about measuring the edit distance per method.

V. Conclusion

In this paper, we analyzed processes of coding to reveal pitfalls for novice programmers. We performed qualitative

analysis by experienced programmers and quantitative analysis based on token-based edit distances. In qualitative analysis, we confirmed two coding patterns that represent pitfalls. We also confirmed the possibility of detecting novices who are facing difficulty by observing transition of the token-based edit distances.

In our future work, we need to continue analysis to find other coding patterns. And also we try to research whether we can support novices or not based on quantitative analysis such as token-based edit distance.

ACKNOWLEDGMENT

This work was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (B) (No.24700030).

REFERENCES

[1] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," in *Proc. the 10th annual SIGCSE conference on Innovation and technology in computer science education* (*ITiCSE '05*), 2005, pp. 14–18.

[2] R. Mosemann and W. Susan, "Navigation and comprehension of programs by novice programmers," in *Proc. the 9th International Workshop on Program Comprehension* (*IWPC '01*), 2001, pp. 79–88.

[3] J.-m. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[4] B. E. Teasley, "The effects of naming style and expertise on program comprehension," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 757–770, 1994.

[5] H. Tamada, A. Ogino, and H. Ueda, "A framework for programming process measurement and compiling error interpretation for novice programmers," in *Proc. 2011 Joint Conference of the 21stInternational Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement* (*IWSM/Mensura 2011*), November 2011, pp. 233–238, nara, Japan.

[6] S. Saito, M. Yamada, H. Igaki, S. Kusumoto, and T. Hoshi, "Programming process visualization for supporting students in programming exercise," in *IEICE Technical Report*, vol. 111, no. 481, March 2012, pp. 61–66, (in Japanese).

[7] "Repository of JavaCC grammars." [Online]. Available: http://java.net/projects/javacc/downloads/directory/contrib/grammars

[8] "JavaCC home." [Online]. Available: http://javacc.java.net/