Generating Fill-in-the-Blank Tests to Detect Understanding Failures of Programming

So Asai, Yoshiharu Yamauchi
Graduate School of Information Science and Engineering
Ritsumeikan University, Shiga, Japan
email:{asai, yamauchi}@de.is.ritsumei.ac.jp

Yusuke Kajiwara, Hiromitsu Shimakawa
College of Information Science and Engineering
Ritsumeikan University, Shiga, Japan
email:{kajiwara, simakawa}@de.is.ritsumei.ac.jp

***Abstract*—This paper proposes a method to generate a fill-in-the-blank test to detect the understanding failures of students in introductory programming course as early as possible. In programming class in educational institutions, what is important for novices is to make them acquire abilities to exert knowledge and skills in programming in an appropriate way according to each situation. The method pays special attention to the fact that students sharing specific understanding failures are likely to write similar inappropriate code. To generate a fill-in-the-blank test, the proposed method determines code fragments to be blanked out in model code, differentiating inappropriate code written by past students from the model code. An experiment has revealed the method can detect students having understanding failure with high precision rates. The fill-in-the-blank tests generated with our method prevent students from leaving their understanding failure unsolved, because teachers can intensively supervise students who fail to acquire abilities to fully exert the knowledge and the skills in the early stages of the programming course.**

***Keywords*—Programming; e-learning; text mining; clustering.**

I. Introduction

In information technology industry, there is a serious lack of workers due to increase of demand for digital products and services [1]. Educational institutions are urged to educate students who have programming skills to resolve the problem.

In educational institutions, students learn how to write program code in a recommended way to exert the knowledge and the skills according to each situation. The goal of programing learning is to make students understand the knowledge and acquire the skills needed to write programs in recommended ways. However, since the instructor teaches all of the students in a uniform manner, some of them have difficulties to immediately understand the recommended ways to write programs. Such students would write slack programs to avoid missing submission deadlines of assignments, neglecting the lecture goals [2]. If the students understanding failure remains unsolved, they will not learn and, if the process continues, they will fail to learn programming.

Teachers are required to detect understanding failures of students as early as possible. To achieve it, they should examine students understanding every class with a test. The way to test the students is at the instructor's discretion. In addition, it imposes a big burden on them, if they have to manually assess answers of many students. Students often submit the same erroneous code as answer to identical assignments in educational institutions. This means, past and current students have identical understanding failures. In this work, we propose a method to detect understanding failures of programming. We adopt a fill-in-the-blank test for the detection of understanding failures, to mitigate the burden on teachers. Fill-in-the-blank tests placing blanks in the part of program where many students are likely to write inappropriate code are expected to be effective to find understanding failure of students. We focus on programs past students wrote to detect code fragments to blank out in fill-in-the-blank tests. Programs past students submitted are represented by vectors figured out in terms of the word similarity to example code and explanations from a textbook. To identify the part involving inappropriate code fragments, the vectors are classified into clusters, to distinguish ones that are the most different from model code. The clusters consisting of pieces of code that are the most different from the model code are referred to as inappropriate clusters. The difference of program code in an inappropriate cluster from the model code is identified to determine code fragments to be blanked out. The method proposed in the work provides an environment to score the fill-in-the-blank tests automatically. It is expected that the fill-in-the-blank tests along with the scoring environment would detect understanding failures of present students without a great effort from the part of the teachers.

In this paper, Section II introduces understanding failures and the efficacy of fill-in-the-blank test on learning programming. Section III explains our method to generate a fill-in-the-blank test. This section illustrates the automatic scoring environment. Section IV indicates the experiment to validate the method with its result. Section V evaluates the result to discuss the validity of the method. Section VI summarizes our work.

II. Understanding Failure of Programming

A. Problems in Programming Class

In the C programming course in educational institutions, students learn programming in each teaching unit, which corresponds to one combination of a lecture class and an exercise class. Students learn knowledge and skills in each teaching unit. To acquire programming ability comprehensively, students are required to solve assignments in a specific teaching unit, using the knowledge and skills they have already learned in the preceding ones. In exercise classes, it is not enough for students only to write a program that behaves in

 ISBN: 978-1-61208-583-8

a required in the assignment. They should train themselves to write the program code in an appropriate manner as they are written in model code. Through the training, they would get abilities to handle the knowledge and skills they have learned. In order to proceed with the training along with the arrangement of units, students must understand all the contents taught in every teaching unit.

Nowadays, various example programs are provided on the Internet and in books [3]. Students try to find example code [2]. If they do not understand the example code well, they would construct program in a cut-and-paste manner, using code fragments extracted from the example code. Such programs involve inappropriate code fragments to implement the required program behavior. They submit executable programs which are composed in the above way, without understanding how to write a program in a recommended way. They should understand each of the code fragments to organize the program as a sequence of code appropriate for the behavior. Even though this behavior can help with the assignment at hand, students should avoid this reckless copying of code fragments without understanding. It might lead the students to another understanding failure in coming classes because they fail to attain sufficient understanding and abilities. Successive understanding failures would drive them to give up programming. Once the students give up programming, it is difficult to direct them again to its learning. Teachers should find students who fail to attain enough understanding in programming classes as early as possible.

B. Appropriate Code and Inappropriate Code

It is required to write programs according to their specifications and a specified programming style [4]. It means there is a recommended way to write a program to be concise and easy to read to accomplish the specific behavior in programming. A program is represented by a sequence of statements meeting a specific pattern.

Students should write the program in a manner model code illustrates to satisfy the requirements in the programming exercises. They can write code similar to the model code, if they sufficiently understand the sequence of statements meeting a specific pattern taught in every teaching unit. This paper refers to a program which implements specified behavior in the recommended way as appropriate code. Appropriate code is represented by a sequence of statements meeting a specific pattern. It becomes similar to the model code in terms of the appearance and the frequency of the statements. On the other hand, code of students who cannot write an appropriate code contains parts different from the model code.

Let us consider two pieces of code for the assignment to understand iteration as shown in Figure 1. Both of them take the same behavior. Code A is described with a for-statement, and code B is described with a while-statement whose condition is true. Code B uses a break statement inside the if-statement to exit from the loop. Although both of them behave as specified, code B is different from the recommended way. In this paper, code which implements specified behavior with a sequence of statements against the recommended pattern is referred to as inappropriate code. It is assumed that inappropriate code is caused by understanding failure of students for correct programming.

```
#include <stdio.h>

int main() {
  int i = 0;
  for ( i = 0; i < 10; i++ ){
    printf("%d\n", i);
  }

  return 0;
}
```

Code A

```
#include <stdio.h>

int main() {
  int i = 0;
  while ( 1 ) {
    printf("%d\n", i);
    i++;
    if ( i >= 10 )
      break;
  }

  return 0;
}
```

Code B

Figure 1. Appropriate and inappropriate code examples

C. Fill-in-the-blank Assignments to Identify Understanding

It is important to evaluate whether students have acquired the knowledge and the skills of programming. Since this evaluation reveals the student achievement, it benefits not only to students but also the teachers, because teachers can plan how to supervise students.

A general way to measure understanding of programming is a scratch test, as which we refer to a test requesting students to write whole program code from scratch. Several patterns of code sequences can construct the program which generates the specified output for the same input. Even if students do not understand the code sequences to be learned, they can meet the requirement by other patterns. In addition, there is a possibility that they have combined code fragments in a reckless manner to generate a program. To assess such a program, the teacher has to read it in order to identify whether they have understood. He needs enormous time and effort because he takes care of many students. Since students proceed with learning of programming based on what they have learned, it is difficult for them to make progress, if they leave understanding failure. The teacher must evaluate the understanding every programming class despite enormous time and effort needed to do so. The teacher should also provide various assignments for students to confirm their understanding. It is not feasible to measure the understanding of programming for many students by scratch tests.

The alternative way to measure programming skills is a fill-in-the-blank test. The teacher blanks out a specific part of the sequence of statements which implements specified behavior, to measure the understanding of a student focusing on a specific point. A few kinds of code are suitable to fill the blank. The blanks are so small that the standard to assess students does not vary with teachers.

In addition, fill-in-the-blank tests are useful to measure program understanding [5]. Appropriate code to implement a

 ISBN: 978-1-61208-583-8

specific behavior consists of a sequence of statements meeting a specific pattern. Suppose a test has a blank hiding a part of a program, so that the blank cannot be filled without knowledge on a sequence of statements matching a recommended pattern. Unless students have programming skills to write the recommended code, they cannot fill the blank with correct code. Such fill-in-the-blank test can measure whether they have learned the programming skills. When they fill some answers to the blank, they read the entire program as well as the code fragments around the blank, to guess the behavior. Since they try to consider a procedure implemented by code around the blank, a proper fill-in-the-blank test makes the students understand how to write the recommended code.

However, when either the place or the size of blanks are irrelevant, it will not detect understanding failures on how to write the recommended code. Thoughtless setting of blanks impedes revealing their understanding failure. A support is necessary for teachers to determine the parts to be blanked out in order to reveal the students who have understanding failure.

D. Related Work

Kashihara et al. [6] generated fill-in-the-blank tests with a program dependence graph to find a blank suitable to measure the understanding of program code. This method makes only one blank in a program. It is considered there are several kinds of inappropriate pieces of code in a program. In particular, the diversity of inappropriate code is prominent in advanced assignments. Many fill-in-the-blank tests must be prepared in advance to detect various understanding failures with the method.

Ariyasu et al. [7] support the automatic generation of fill-in-the-blank tests meeting intention of teachers with syntactic analysis of program code. However, the teachers themselves must look over the understanding status of all the students to determine what should be examined in the test. It depends on the abilities of the teacher to generate fill-in-the-blank tests which can reveal programming understanding. These works take into account neither of latent understanding failure nor supports to determine what should be examined.

Funabuki et al. [8] proposed a Java learning system to generate a fill-in-the-blank assignment function which assists learning of reserved words. This system blanks out in the model code by selecting reserved words randomly. It is difficult to measure understanding failure for all of programming skills to be acquired since teaching units other than reserved words are not covered. An alternative method is necessary to generate fill-in-the-blank tests to solve these problems.

III. Revealing Understanding Failure from Past Student Code

A. Goal and Significance

To complete a fill-in-the-blank test, students have to determine a code fragment to fill in a blank after they understand a sequence of code around the blank. It forces them to understand the meaning of the code fragments before and after the blank. An automatic scoring system, which allows the students to retry the test many times, allows them to become aware of their own understanding failures.

Since any fill-in-the-blank assignment is issued based on a model code, programming novices can study how to organize a program with code fragments appropriately to achieve the required behavior of the program. For each assignment issued in previous programming courses, the teacher has accumulated examples of inappropriate code many of the past students in the educational institution wrote. Understanding failures seem to vary with educational institutions, because students with similar understanding belong to a specific educational institution. Teachers should not issue universal assignments presented in commercial textbooks. Using the information from past students, teachers can generate fill-in-the-blank tests suitable to detect understanding failures specific to the educational institution. Furthermore, the automatic scoring system releases teachers from boring tasks to examine a lot of program code of the students. The teachers can concentrate on the intellectual work to generate fill-in-the-blank tests to identify students who have understanding failures.

B. Inappropriate code of Past Students

In this work, we propose a method to generate fill-in-the-blank tests to identify understanding failures which cause students to write inappropriate code. Figure 2 shows the method overview. In a specific educational institution, students who have identical understanding failures have a tendency to write the same kind of inappropriate code over the years. Using this tendency, the method generates fill-in-the-blank assignments from program code past students wrote. It is assumed that the same assignments are given in the past and the present years. Present students are likely to write almost the same inappropriate code as past students did, if both of them have identical understanding failures. It is possible to find a code fragment to be blanked, if we reveal inappropriate code fragments past students wrote.

In order to find inappropriate code fragments, we classify programs written by past students. The similarity of a program to example code in each teaching unit is calculated based on the appearance frequency of statements such as if-statements and for-statements. Every program code is represented by a vector in a coordinate space. Each element of the vector corresponds to a teaching unit. Programs are classified into clusters in terms of the similarity to example code in the teaching units. Student code submitted for an assignment is regarded as inappropriate, if it is very different from a model code of the assignment.

It is conceivable that there are several inappropriate fragments in a program. To determine which parts of code are inappropriate, the method computed the difference between student code and the model code. This method reveals parts

 ISBN: 978-1-61208-583-8

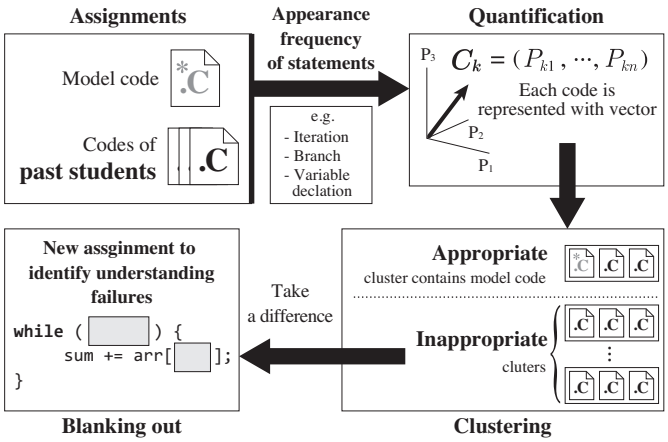

Figure 2. Method overview

involving inappropriate code. The parts express the code fragments to be blanked out.

Fill-in-the-blank tests generated by the method reveal whether each of current students understands the blanked-out parts. Students who cannot fill the blanked-out parts correctly are likely to have a specific kind of understanding failure. Teachers should supervise those students, to prevent them from leaving the understanding failure for any teaching units unsolved.

C. Extraction of Characteristics with Text Classifier

For each assignment, a teacher would prepare a model program to show students how to write an appropriate code in a recommended way as it is written in a textbook like [9]. The proposed method represents quantitatively how student code is appropriate like the model code.

Novice programmers of conventional programming languages such as C learn programming skills according to teaching units such as iteration and arrays. The characteristics of example code and sentences to explain them vary with teaching units in a textbook for C novice programmers. A program similar to example and their explanation in a teaching unit is regarded to be in accordance with what are learned there. For each assignment, the proposed method treats its model code and student code as source code to calculate their similarity to the teaching units in the textbook. A program is represented by a set of the probability that the program is similar to every teaching unit in the textbook.

Program code and comments in a program as well as example code and sentences in the textbook are divided into words with MeCab [10], which is a tool for morphological analysis. The text classifier implemented with the Naive Bayes filtering [11] records of the appearance degree of words in every teaching unit of the textbook in advance. For a program, the text classifier calculates the appearance degree of words used in every teaching unit of the textbook. The appearance degree probabilistically represents to which teaching unit of the textbook the source code is similar in terms of the

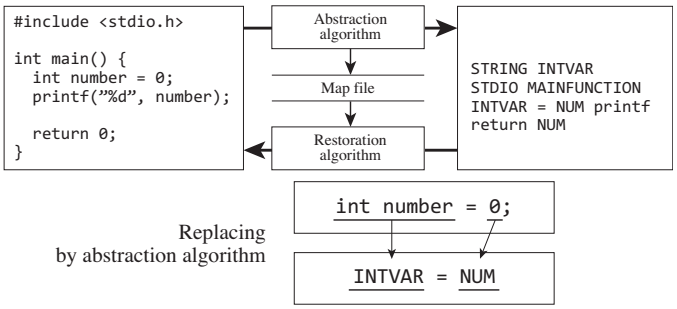

Figure 3. Translation to abstracted code

appearance frequency of the words. First, the proposed method calculates the appearance degree representing the similarity to every teaching unit. It selects a specific number of teaching units of high similarity. To address characteristics of source code, the proposed method uses a vector whose elements correspond to those teaching units. Second, the method represents characteristics of source code written by past students with vectors having those elements. The vectors of student programs obedient with the teaching units would be similar to that of the model code.

Let us calculate the characteristics of programs, including model code and student code. To represent the characteristics of programs, it is preferable to eliminate the variance of identifier names and literal values in every program. To achieve it, the method translates source code into abstracted code. Figure 3 shows an example of the abstraction. In the figure, integer variables, numerical values, and for-loop statements are replaced with INTVAR, NUM, and FOR, respectively. The abstraction makes it explicit which statements are used frequently in source code. Although this abstraction causes the order of words to permute, it does not matter because the method calculates only the appearance frequency of words. Using correspondence of replacing words to original code, the abstracted code can be restored to the original code.

The vectors of programs past students submit for an assignment is compared with that of the model program for the assignment. The vectors of programs, which use appropriate code like the model program are placed near the vector of the model program. Inappropriate code involves eccentric code sequences or redundant processing, which are not found in the model code. It is considered the vector of a program composed of them is considerably far from the vector of the model program. The method classifies the programs written by past students into 2 categories; one is appropriate like a model program, while the other is inappropriate.

D. Classifying into Appropriateness and Inappropriateness

In this work, it is assumed that both of past and present students who have a same understanding failure write a similar inappropriate code.

The proposed method classifies student program and the model program for an assignment to find inappropriate code the students are likely to write. The method applies a cluster

 ISBN: 978-1-61208-583-8

analysis for vectors which represent student programs and the model program. The number of clusters cannot be determined in advance, due to the variance of inappropriate code for every assignment. The method adopts the Ward's method for the clustering. At first, each assignment is assumed to have less than three kinds of inappropriate code. Using a dendrogram generated as a result of the analysis, the method partitions clusters so that code which seems to be inappropriate should not be placed in the same cluster as the model code. The method refers to the cluster containing the model code as an appropriate cluster, while the other cluster as an inappropriate cluster.

Progmras classified into appropriate cluster have high similarity with the model program. Students whose code in the appropriate cluster are considered to write their code, after they understand knowledge and skills to be acquired. Meanwhile, programs classified into inappropriate clusters contain some inappropriate code fragments. Students who wrote the programs in the inappropriate clusters seem to be accompanied with understanding failures corresponding to the inappropriate code. Those students should be supervised as early as possible not to leave their understanding failures uncorrected.

E. Blanking Out Code from Differences

If students write programs falling into an inappropriate cluster, teachers should notify the students that they may have a specific understanding failure. The students with the understanding failure should understand why the model code has code fragments different from theirs. Since it leads them to the right understanding, they could modify their program closer to the model program.

A filling-in-the-blank test is generated to examine whether the students who wrote programs in an inappropriate cluster commit the specific understanding failure. A program in the inappropriate cluster has a code fragment where the usage of statements in programming is different from that in the model program. A filling-in-the-blank test is generated with a part of the code fragment blanked out.

The proposed method has to identify what part is to be blanked out in the model code. All programs in an inappropriate cluster have same tendency in its usage of programming statements. One of programs in the inappropriate cluster is picked up, to take its difference from the model code. As it is pointed out in Section III-C, variance of identifiers is inconvenient to figure out the difference. Abstracted code made from the inappropriate code is used to take differences, in the same way as in the classification. When we use the abstracted code, the difference does not come from variance of identifier names, but from discordance of statement usages in C programming. In the model code, the proposed method detects the code fragments corresponding to the difference. In case of Figure 4, a part of the iteration statement in the model code is replaced with a blank, because there is difference in the while statements between the two programs. For example, the condition of the while-statement should be blanked out

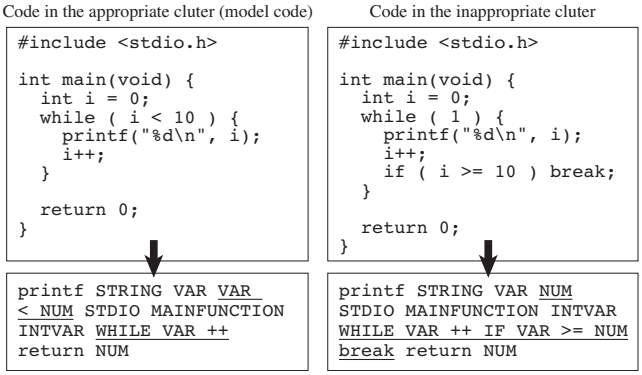

Figure 4. Difference between appropriate and inappropriate code

in the model code. Students having understanding failure for the iteration with while-statement cannot fill the blank correctly, because they always try to use an infinite loop. The above procedure is applied to all the inappropriate clusters to determine blanks suitable to detect understanding failure corresponding to each of the inappropriate clusters.

A blank in a filling-in-the-blank test implies the part where the students might write inappropriate code. Suppose they cannot fill one blank, while they can fill appropriate code for other blanks. It means they have a specific kind of understanding failure regarding to the code fragment around the blank. Providing students with various filling-in-the-blank tests, we can identify students committing every kind of understanding failure. Teachers should supervise the students to understand not only the code to fill the blank, but also why they should write the code, because they wrote the inappropriate code based on a wrong way of understanding.

F. Automatic Scoring for Fill-in-the-Blank Tests

It is important that fill-in-the-blank tests have the students consider what a correct answer is. Various fill-in-the-blank tests given to students clarify understanding failures of students. The proposed system provides an automatic scoring system of fill-in-the-blank tests, so that students may check the correctness of their answers at any time. This system is executed on Web server. Students engage in fill-in-the-blank tests on Web pages. When a student submits a fill-in-the-blank test filling its blanks with his code, the system scores it to notify him the result immediately. The immediate notification makes students strongly conscious of the tests they fail. It prevents the students from leaving themselves unaware of the understanding failures.

Conventional ways to automatic scoring of filling-in-the-blank tests mainly use string comparison of student's answers with ones teachers expect. It judges that the student's answers are correct when they are fully matched with the expected ones. Students send diverse answers with trivial differences. For example, when several variables are initialized with literals, their order does not matter. Since correct answers exist infinitely, the string comparison is unsuitable for the automatic scoring.

 ISBN: 978-1-61208-583-8

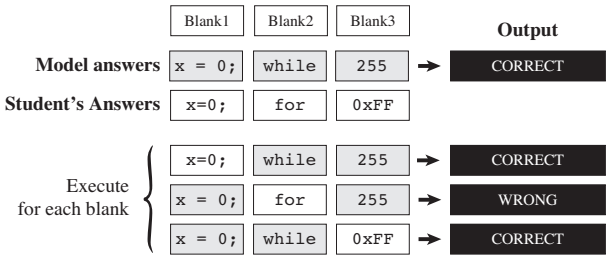

Figure 5. Working example of the automatic scoring system

The proposed method in this work adopts an alternative way which compares execution results of programs to address the diverse answers. Programming assignments for novices would require students to write a program with some outputs on CUI. The way of automatic scoring in the proposed method generates an executable code using both of strings a student answers and ones a teacher prepares. Only one of the blanks is filled with the student answer corresponding to it, while the others with the ones the teacher prepares. The filled code is executed to verify whether the execution output is the same as the output of the code filled with all of the strings the teacher prepares. It enables each of student's answers to be separately scored, even if a filling-in-the-blank test has multiple blanks.

Let us explain the scoring way with a fill-in-the-blank test with 3 blanks shown in Figure 5. Let the prepared answers for Blank1, Blank2, and Blank3 be code fragments "x = 0", "while", and "255", respectively. When they are specified for the blanks, the output is expected to be string "CORRECT", which is printed on CUI. Suppose a student specify code fragments "x = 0", "for", and "0xFF" as his answers.

First, Blank1 is filled with student's answer "x = 0", while the other blanks with the prepared answers. This program code prints string "CORRECT" when it is executed. Since the output is the same as the expected one, the student's answer is judged to be correct. Second, Blank2 is replaced with student's answer "for", while Blank1 and Blank3 are replaced with their prepared answers. The execution result of this program code is different from the expected one. It turns out student's answer "for" is wrong. Finally, Blank3 is filled with student's answer " 0xFF" . Suppose "CORRECT" is printed as a result of the execution. Although the student specifies a code fragment different from the prepared one for Blank3, he is judged to answer correctly, because the output is identical with the expected one. The student is notified that he presents correct answers for both of Blank1 and Blank3, while fails for Blank2.

IV. Experiment

A. Method and Purpose

We conducted an experiment to verify that the fill-in-the-blank tests generated by the proposed method can detect understanding failures. 122 students at College of Information Science and Engineering of Ritsumeikan University participated in the experiment. All of them attended an introductory C programming course, which consists of lectures and exercises, over a year ago.

In the experiment, each student answered the assignments with 3 phases. In the first phase, the student wrote a program from scratch. Let us refer to it as a scratch test. In the second and the third phases, the student solved the two types of fill-in-the-blank tests. One type of the fill-in-the-blank tests is generated with the method, and the other is produced manually by a teacher. The students repeated the answering for these fill-in-the-blank assignments. For the former type of the fill-in-the-blank assignment, the fill-in-the-blank tests were produced, blanking out several code fragments of its model code. To prevent the students from reusing code fragments, variable names in the model code differ in the two fill-in-the-blank tests. For the latter type, when a teacher produced fill-in-the-blank tests, he was provided with the assignment and its model code. He specified code fragments to be blanked out in the model code, as well as his intention regarding what he wanted to confirm with the blanking.

They signed in a website for the experiment with personal user ID and password to challenge the tests. To make students engage in programming assignments as usual, the students were permitted to carry out the tests in any environment such as home and in the university. We settled 2 weeks for the experiments. There was no time constraint other than the submission deadline. Since we intend that they solved the tests for themselves, code copied from websites and digital books should be excluded from the experiment data as cheatings. In order to find the cheating, we examined rapid character filling for blanks. In the scratch test, students can compile their programs at any time to check the output. In the fill-in-the-blanks tests, they could check whether the answers are correct, submitting their answers to the automatic scoring system. The system notified them the scoring result immediately. When they finished or gave up trying a test, they pressed the button to move to the next test.

B. Generating Fill-in-the-Blank Tests

We prepared fill-in-the-blank tests from past programming exercises based on the method. Among assignments given in C Programming Exercise Courses, which College of Information Science and Engineering of Ritsumeikan University offered in the first semester of 2016 academic year, the following five ones are chosen as the tests for the experiment. They are related to knowledge and skills for which students are likely to have understanding failure, such as iterations, functions, arrays, and pointers [12]. We refer to each of the followings as Test A to Test E, respectively. The parentheses indicate teaching units to which the problem relates.

Test A (iteration) Calculate to print interior angles of each regular N-sided polygon for the integer N from 3 to 12 with a while-statement. When the interior angle is not an integer, skip the iteration with a continue-statement.

Test B (function and array) Print the number of characters from A to Z included in any character strings specified from the standard input.

 ISBN: 978-1-61208-583-8

Test C (iteration) Let us build up a pyramid, placing cube stones without gaps. Given the number of cube stones, figure out the number of pyramid steps and remaining stones.

Test D (function and array) Implement a function to make product of two-by-three matrix and a 3-dimensional vector. Print the product for element values given from the standard input.

Test E (function, pointer and iteration) Inserting a hyphen before non-vowel characters in any character strings given from the standard input, print the string which has the same length as the original one. Use the given function to judge whether a character is a vowel.

For each of the assignments, a teacher wrote a model code, while students submitted their programs. To generate a fill-in-the-blank test for the assignment, the proposed method classifies vectors representing the model program and past student programs.

In order to generate a fill-in-the-blank test, the method picked up a program from the inappropriate clusters, respectively, to take its difference from the model code. On the other hand, the fill-in-the-blank tests were produced by a teacher who engaged in the class of C Programming Exercise. After he determined the intention of each test, he blanked out several parts in the model code, referring to assignments and their model programs.

C. Scoring Results

We obtained answers for the tests from the 122 subjects. Some of the answers were not finished solving tests completely or suspected of the cheatings. We do not use such invalid answers for evaluation of the proposed method. To ensure the fairness in the experiment, we used answers that each subject submitted for the first time. We score all the scratch tests by hand to find inappropriate code fragments. We judged answers for the scratch tests incorrect, if we find inappropriate code in them. The followings are inappropriate code fragments found in the scratch tests. For example, A1 and A2 are found for Test A.

A1 The place to do increment for the loop.
A2 The condition to call continue statement.
B1 The place to call the function toupper.
B2 Counting the number of each alphabetic character.
C1 The initial value to count the steps.
D1 Not initializing an array to store the matrix product.
D2 Not using for-statement for the matrix products.
E1 The direction to search character strings.
E2 Escaping from the loop with break statement.

Table I shows the rate of correct answers which the subjects have given finally in the scratch tests and the both kinds of fill-in-the-blank tests from the proposed method and the teacher production. The rate is calculated within the valid answers. In the scratch tests, correct code means an executable program working normally without any inappropriate code. The values in parentheses indicate the rate difference of each fill-in-the-blank test against the scratch test.

TABLE I. RATE OF CORRECT ANSWERS

TEST	SCRATCH	METHOD	TEACHER
A	0.87	0.92 (0.05)	0.94 (0.07)
B	0.29	0.83 (0.54)	0.83 (0.54)
C	0.60	0.60 (0.00)	0.92 (0.32)
D	0.61	0.74 (0.13)	0.87 (0.26)
E	0.20	0.70 (0.50)	0.31 (0.11)

TABLE II. CONDITIONAL PROBABILITIES

CLUSTER	$n(B)$	$n(S)$	$n(S \cap B)$	$P(B \mid S)$	$P(S \mid B)$
A1	11	14	9	0.64	0.81
A2	8	8	4	0.50	0.50
B1	37	70	28	0.45	0.75
B2	23	27	17	0.63	0.73
C1	40	38	25	0.66	0.63
D1	30	36	18	0.50	0.60
D2	24	35	19	0.54	0.79
E1	14	17	14	0.82	1.00
E2	11	10	8	0.80	0.73

D. Conditional Probabilities

In the experiment, we focus on the subjects whose answers were incorrect for either the scratch tests or the fill-in-the-blank tests. A conditional probability is defined in order to prove relevance of fails in the fill-in-the-blank tests generated by the proposed method with incorrect answers in the scratch tests. $P(Y \mid X)$, the probability of Y under the condition X, is given by the formula:

$$P(Y \mid X) = \frac{n(X \cap Y)}{n(X)} \quad (1)$$

Where $n(X)$ denotes the number of event X. We let S and B represent, respectively, an event where the scratch test is incorrect, and a fail in the fill-in-the-blank test. Table II shows the conditional probabilities.

V. EVALUATION AND DISCUSSION

A. Causing Inappropriate Code Fragments

Table II shows the conditional probabilities students write inappropriate code for the scratch tests and the fill-in-the-blank tests generated by the method.

$P(B \mid S)$ indicates the probability a subject who writes a specific inappropriate code in the scratch test mistakes in the filling-in-the-blank test presenting a blank he might fill with the same kind of inappropriate code. It is the recall, which implies to what rate fill-in-the-blank test can detect subjects who has understanding failure. On the other hand, $P(S \mid B)$ indicates that a subject who have mistaken in the blank of the fill-in-the-blank test writes the same kind of inappropriate code in the scratch test. It is the precision, which implies how much we can trust results of fill-in-the-blank tests generated by the method.

From the table, $P(S \mid B)$ shows the high values over 0.70 in the six inappropriate clusters, A1, B1, B2, D2, E1, and E2 of the fifteen. Meanwhile, $P(B \mid S)$ was more than 0.70 only in the two items. It suggests students giving the wrong answer in fill-in-the-blank tests are likely to write inappropriate code fragments in the corresponding scratch tests. That suggests the teacher may conclude that they have the understanding

 ISBN: 978-1-61208-583-8

failure for the code fragments to be filled in the test. The teacher should supervise them to correct their understanding failure. The recall rates, which are not high enough, mean fill-in-the-blank tests generated by the method fail to detect many students who are likely to write the inappropriate code. We need to deal with this problem.

Since the proposed method automatically scores fill-in-the-blank test as explained in Section III-F, it does not require heavy effort of the teacher to score. They can give students more fill-in-the-blank tests than scratch tests. In the experiment, we gave only one test for each kind of inappropriate code. To achieve specific behavior of a program, students need to acquire programming knowledge or skills corresponding to it. When students commit understanding failure for the knowledge or the skills, they are likely to write inappropriate code to achieve the program behavior. Even in other assignments, they would use the same kind of inappropriate code to write code founding on the identical programming skill. The teacher can detect more students with the understanding failure who write the inappropriate code, giving several fill-in-the-blank tests to examine the identical programming knowledge or skills.

B. Comparison with Correct Answer Rates

In the experiment, students engaged in two kinds of fill-in-the-blank tests; one was generated by the proposed method while the other produced by the teacher. Let us compare the rate of correct answers for the two kinds of fill-in-the-blank tests.

Fill-in-the-blank tests are required to detect students who write inappropriate code caused by understanding failure. Suppose a fill-in-the-blank test which places blanks in any part where the students might write inappropriate code for the assignment. It can detect any kind of understanding failure which might occur for the assignment. Such fill-in-the-blank test has a feature to detect the occurrence of every inappropriate code in the scratch test. In other words, it is preferable that the rate of correct answers of the fill-in-the-blank test approaches to that of the scratch test.

Let us compare the differences of the rate of correct answers for the fill-in-the-blank test from the rate of the scratch test, in both cases: the ones generated by the proposed method and the ones produced by the teacher. See Table I. For Test A, C and D, the method has smaller difference than the teacher. For Test B, the difference is identical each other, while the teacher has smaller difference for Test E. For four tests of the five, the proposed method has similar correct answer rate to the scratch test. For Test E, it is considered that difficulty of the test is advanced and its program is not suitable for blanking out. It is necessary to select basic assignments which contain the teaching units to be acquired.

The proposed method assumed students in an educational institution would commit same understanding failures for same assignments over years. Under this assumption, the method analyzes programs of the past students in the same educational institution quantitatively, to identify where inappropriate code frequently occur in the model code for each assignment. On the other hand, the teachers blanked out parts of the model code without quantitative analysis. Due to the quantitative analysis, fill-in-the-blank tests by the method were more successful to find the students who has understanding failure than the ones by the teacher.

VI. Conclusion

In this paper, we have proposed the method to generate a fill-in-the-blank test which detects understanding failure of present students. The method classifies programs past students wrote, to find inappropriate code fragments caused by the understanding failures. Scoring results of our fill-in-the-blank test contribute to finding students who would be at loss due to unconscious understanding failure early.

The high precision values from the experiment indicate that our fill-in-the-blank tests can reveal students having understanding failure for knowledge and skills of programming from inappropriate writing ways of code. The method enables teachers to supervise students poor in understanding intensively.

In the future, we give more fill-in-the-blank tests to detect students who have understanding failures completely. In addition, it is necessary to discuss concrete teaching ways suitable for each kind of understanding failure, analyzing inappropriate code caused by it.

References

[1] Ministry of Economy, Trade and Industry, "Summary of research results on latest trends and future estimates of it talent," http://www.meti.go.jp/press/2016/06/20160610002/20160610002.pdf [retrieved: December, 2016].

[2] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *ISESE '04 Proceedings*, 2004, pp. 83 – 92.

[3] Stack Exchange, Inc., "Stack overflow," http://stackoverflow.com [retrieved: January, 2017].

[4] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style 2nd Edition*. McGraw Hill, 1978.

[5] A. Kashihara, M. Soga, and J. Toyoda, "A support for program understanding with fill-in-blank problems," *JSISE*, vol. 15, no. 3, pp. 129 – 138, 1998, in Japanese.

[6] A. Kashihara, A. Terai, and J. Toyoda, "Making fill-in-blank program problems for learning algorithm," in *ICCC'99*, 2001, pp. 776 – 783.

[7] K. Ariyasu, E. Ikeda, T. Okamoto, T. Kunishima, and K. Yokota, "Automatic generation of fill-in-the-blank exercises in adaptive c language learning system," in *DEIM Forum*, 2009, pp. 776 – 783, in Japanese.

[8] N. Funabiki, Y. Korenaga, T. Nakanishi, and K. Watanabe, "An extension of fill-in-the-blank problem function in java programming learning assistant system," in *2013 IEEE Region 10 Humanitarian Technology Conference*, Aug 2013, pp. 85 – 90.

[9] M. Moriguchi, "Worried C," http://9cguide.appspot.com/en/index.html [retrieved: December, 2016].

[10] T. Kudo, K. Yamamoto, and Y. Matsumoto, "Applying conditional random fields to japanese morphological analysis," in *Proceedings of EMNLP*, 2004, pp. 230 – 237.

[11] F. Pedregosa et al., "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825 – 2830, 2011.

[12] L. Kaczmarczyk, E. Petrick, J. P. East, and G. L. Herman, "Identifying student misconceptions of programming," in *Proceedings of SIGCSE'10*, 2010, pp. 107 – 111.

 ISBN: 978-1-61208-583-8