Detecting Exploratory Programming Behaviors for Introductory Programming Exercises

Erina Makihara*, Hiroshi Igaki†, Norihiro Yoshida‡, Kenji Fujiwara* and Hajimu Iida*

*Nara Institute of Science and Technology, Japan
Email: {makihara.erina.lx0@is, kenji-f@is, iida@itc}.naist.jp
†Osaka Institute of Technology, Japan
Email: hiroshi.igaki@oit.ac.jp
‡Nagoya University, Japan
Email: yoshida@ertl.jp

Abstract—Developers often perform the repeating cycle of implementation and evaluation when they need to deal with the unfamiliar portion of the source code. This cycle is named as exploratory programming. We regard exploratory programming as an effective way not only to improve novice’s programming skill but also to support educators in programming exercise in University. Because when novices often use the exploratory programming, it means novices struggle to solve their assignments. Therefore, educators should grasp which elements, APIs or blocks novices often used exploratory programming for. In this paper, firstly we propose the definition of novice’s exploratory programming to collect logs of exploratory based on various granularity by novices. Secondly, we propose an algorithm based on our proposed definition to automatically detect exploratory programming behaviors. We also conducted a small case study. As a result of automatic detection, our proposed algorithm allows us to know what elements of program novices often feel difficult and struggle for.

I. INTRODUCTION

In software development, when developers use unfamiliar programming languages, algorithms and APIs, it is considered desirable to make multiple variations of software iteratively [2], [3]. Such iterative programming cycle of edit, compile, run, and test is called exploratory programming [4].

Several conventional research indicates exploratory programming contributes to improving the quality of software design and its implementation [5], [6], [7]. Therefore, multiple tools and methodologies are proposed to support skilled software developers to perform exploratory programming [8], [9].

In precedence research, we proposed a programming environment Pockets to aim at supporting the exploratory programming for novices [1]. Pockets visualizes the programming process as a sequence of thumbnails, which shows a snapshot, a timestamp and an execution result when novices perform some actions such as save, compile and run as shown in Figure 1. Furthermore, by using Pockets, novices can easily revert to an earlier revision (That is called “backtrack [8]”) by clicking the corresponding thumbnail. Through a case study of Pockets in our precedence research, we confirmed that our Pockets supports novices to backtrack with exploratory programming. Through the histories of Pockets, we also confirmed that educators gain better understanding know how novices perform exploratory programming. Such histories tell which part of the source code novices regarded as difficult in programming exercise concretely. However, it is hard for educators to grasp precisely when and where novices performed exploratory programming only from the histories of Pockets in a short term.

Therefore, in our research, we aim to support automatic identification of exploratory programming behaviors by novices. We define “exploratory programming by novices” if novices edit the same block, same line or same element in a line in the adjacent revisions. Some programming environments like Pockets can record revisions fine-grained [10], [11]. Based on the definition, our proposed algorithm identify the exploratory programming by novices with various granularities (such as a block, a line, or an element in a line) from the fine-grained revisions obtained by such environment.

We organize the rest of this paper as follows. We state the detail of our definition and the step of automatic detection in Section II. Next, the conducted small case study is discussed in Section III. Finally, we summarize this paper with future work in Section IV.
II. Approach and Its Uniqueness

In order to detect exploratory programming behaviors by novices, we firstly define the exploratory programming by novices. Afterwards, we introduce the detection algorithm.

A. The definition of exploratory programming by novices

In our previous study, we investigated the actual exploratory programming by novices in an introductory programming exercise class for college student [12]. As the result of our investigation, we identify that novices perform exploratory programming of various granularities, for example, in the same block, the same line or the same element in a line. Therefore, we define “exploratory programming by novices” if novices edit the same block, the same line or the same element in a line between the adjacent revisions. Here, we assume that the revision is recorded by the programming environment every time when the user compile or run. In several programming languages such as Java and C, a block is a portion of the source code which is surrounded by curly braces. In our definition, a block includes expressions or statements characterizing the block in addition to the portion surrounded by curly braces. For example, in Figure 2, the statement “for(i = 0 ; i < 5 ; i++)” is contained in the “for” block.

In this paper, we estimated C or Java as the programming language used in introductory programming exercise, and both of languages use curly braces when a programmer writes conditional statement or loop. However, other programming languages which do not use the curly brace (e.g., python, Ruby) also have block definition. Therefore, we think block definition we proposed is able to extend for such programming languages.

In the source code, blocks are usually nested. We also define the block depth expressing nested level of the block. In our definition, block depth indicates the nested level from the outside of the program. We show an example of block depth in Figure 2. In this case, the block depth for each block is as follows:

Depth 0: main (line: 1)
Depth 1: for (line: 4)
Depth 2: if (line: 5)
Depth 3: if (line: 6)

In addition, when novices repeatedly modify the source code over the nested blocks, the modification are regarded as exploratory programming in the block with the least block depth. For example, if the conditional statement (depth 2/line: 5) and conditional statement (depth 3/line: 6) are edited between the adjacent revisions, those edits are regarded as exploratory programming in the conditional statement (depth 2/line: 5), because it is the block of the least depth which includes all the edited blocks.

Our research uniqueness is that our proposed definition of exploratory programming covers multiple granularities of exploratory programming. By analyzing exploratory programming based on various defined granularities, educators can specify the block depth and the granularity to investigate the trend of novice’s exploratory programming behavior and support novices more efficiently.

B. Algorithm for automatic detection

To detect the exploratory programming by novices automatically, we propose the algorithm as shown in Figure 3, and the steps are as follows.

**STEP1** Specify 2 adjacent revisions.

**STEP2** Compare and identify the diff between two adjacent revisions.

In Figure 2, the source code which includes some block depth

```c
01: int main(void){
03:    int i;
04:    for(i = 0 ; i < 5 ; i++){
05:        if(data[i]%4 == 0){
06:            if(data[i]%100 == 0 && data[i]%400 == 0 ){
07:                printf("%d is leap year\n",data[i]);
08:            }
09:        }
10:        printf("%d is not leap year\n",data[i]);
11:    }
12: }
```

Fig. 2. The source code which includes some block depth
diff-match-patch\(^1\) library for identifying the diff as the edited part of the source code.

**STEP3** Tokenizing the source code of STEP2, and then investigating the granularity of the edited part. The granularity of the edited part is classified as follows:

- **Granularity MIN**
  - Single element (e.g., operator, a variable, an identifier, a literal)
- **Granularity MID**
  - One line
- **Granularity MAX**
  - Multiple lines

The component with the minimum granularity is preferentially chosen.

STEP1, 2 and 3 are repeated for each adjacent revision pair. As a result from STEP1 to STEP3, the necessary information of all blocks which include edited part of the source code between revisions is detected. Such information contains (1) depth of the block, (2) line number of the block, (3) name of the block, (4) edit type, (5) line number of edited part, (6) the additional contents and (7) the deleted contents. Besides, (4) the edit type indicates any action of addition, deletion or replace.

### III. Case study

To investigate how our automatic detection algorithm works adequately for the exploratory programming by novices, we conducted a small case study.

**A. Target data**

The outline of the target is shown in Table I. In the introductory programming exercise course, 33 students solved an assignment (Figure 2) using Pockets. Pockets recorded revisions every time students compile or run. Then, Pockets collects the student ID, edit time, a snapshot of the source code, compile or runtime error messages for each revision and stores them in the database.

<table>
<thead>
<tr>
<th>Number of students</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming language</td>
<td>C</td>
</tr>
<tr>
<td>Total revision number</td>
<td>510</td>
</tr>
<tr>
<td>The contents of assignment</td>
<td>Check each numbers in the array is leap year or not. Conditions: (A) Leap year is divisible by 4 (B) Leap year is not divisible by 100 (C) Leap year is divisible by 400</td>
</tr>
</tbody>
</table>

| Table I: An outline of case study |
|-----------------|---|
| **Number of students** | 33 |
| **Programming language** | C |
| **Total revision number** | 510 |
| **The contents of assignment** | Check each numbers in the array is leap year or not. Conditions: (A) Leap year is divisible by 4 (B) Leap year is not divisible by 100 (C) Leap year is divisible by 400 |

### III. Case study

As stated above, the retrieved exploratory programming behavior in introductory exercise to detect exploratory programming behavior at several level of granularity. Therefore, all such revisions are not included in our investigation.

According to the result of this case study, the largest proportion of granularity of exploratory programming is element of line. The judgement of leap year in this assignment can be solved by two ways, one is written in one line by using logical operators, another is written with a few lines by nesting conditional statements.

An example of detected exploratory programming by the proposed automatic detection is shown in Figure 4 as the corresponding edit history. This student edited the source code of Figure 2 from (a) to (d) in chronological order. Though these revisions did not include any compile or runtime errors, the output of the source code about data[3](2100) wrong. Therefore, he tried debugging by using printf at (b). As a result, he found there were some problems in the conditional expression of if, and then he edited source code correctly. In this case, he performed exploratory programming to solve the assignment because of the lack of understanding on the algorithm for detecting leap year, not the grammar of conditional statements. Hence, educators should explain to him, not the grammar of conditional statements but the algorithm for solving this assignment.

Besides, in this assignment, many students struggle to write conditional expression in the if block. As shown in Table II, many students performed element level exploratory programming in the if block. In our further analysis, we found that many students misused bool operators and comparison operators in the assignment. Based on the detected result, educators should explain about if-statement including usage of bool operators and comparison operators.

As stated above, the retrieved exploratory programming information by using Pockets help educators to give more accurate advice or feedback to each student. By summarizing and visualizing exploratory programming information of the whole introductory programming exercise, educators can consider which specific topic should be explained more in detail in the next class conveniently. The overview of other supporting method examples are in Figure 5.

### IV. Summary and future work

In this paper, we firstly introduced a refined definition of exploratory programming for introductory exercise to detect exploratory programming behavior at several level of granularity. Afterwards, we presented an algorithm for the detection of exploratory programming behavior in introductory

### TABLE II: The result of case study

<table>
<thead>
<tr>
<th>Edit granularity</th>
<th>MAX(block)</th>
<th>MIN(element)</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>main</td>
<td>for</td>
<td>if</td>
</tr>
<tr>
<td>MAX(block)</td>
<td>4</td>
<td>70</td>
<td>87</td>
</tr>
<tr>
<td>MIN(line)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MIN(element)</td>
<td>48</td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td>TOTAL</td>
<td>52</td>
<td>136</td>
<td>149</td>
</tr>
</tbody>
</table>

\(^{1}\)https://code.google.com/archive/p/google-diff-match-patch/
programming exercises. As the result of the case study, we found that the lack of the understanding of algorithm and programming language grammar were considerable factors that induce novices to perform exploratory programming behavior.

As a future work, we plan to identify the trend of novices exploratory programming by applying data of large-scale collected for Pockets and then propose a learning environment that supports novices in introductory programming exercises.

Fig. 4. Detected exploratory programming

(a) Revision.1

```c
05: if(data[i]%4 == 0){
06:   if(data[i]%100 != 0){
07:     if(data[i]%400 == 0 ){
08:       printf("%d is leap year
"), data[i]);
09:     } }  
10:  }
11:  }
12: }
```

(b) Revision.2

```c
05: if(data[i]%4 == 0) {
06:     if(data[i]%100 != 0) {
07:         if(data[i]%400 == 0) {
08:             printf("%d is leap year
"), data[3]);
09:         } }  
10:     }
11:  }
12: }
```

(c) Revision.3

```c
05: if(data[i]%4 == 0 &&
    data[i]%100 != 0 &&
    data[i]%400 == 0 ){
06:     printf("%d is leap year
"), data[3]);
07:  }
08: }
```

(d) Revision.4

```c
05: if(data[i]%4 == 0 &&
    data[i]%100 != 0 &&
    data[i]%400 == 0 ){
06:     printf("%d is leap year
"), data[3]);
07:     }
08: else{
09:     printf("%d is not leap year
"), data[i]);
10:     }
```

Fig. 5. An example of an approach to support exploratory programming based on the collected data

REFERENCES


