A Tool to Suggest Similar Program Element Modifications

Yujiang Yang
Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
yujiang.yang@fuji.waseda.jp

Kazunori Sakamoto
National Institute of Informatics
Tokyo, Japan
exkazuu@nii.ac.jp

Hironori Washizaki, and Yoshiaki Fukazawa
Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
{washizaki, fukazawa}@waseda.jp

***Abstract*—Many program tasks require continuous modification of similar program elements, which is burdensome on programmers because continuous modifications are time consuming and some modifications are easily overlooked. To resolve this issue, we extracted all possible matching elements via similarity patterns from recently modified elements using a sub syntax tree comparison and then created a tool, SimilarHighlight. Our tool suggests similar program elements that may be modified during the next modification. Potential elements are highlighted and their text can be immediately selected by shortcut keys. Evaluations indicate that SimilarHighlight can improve programming productivity. Currently, our tool supports C, C#, JAVA, JavaScript, and PHP, but in the future we will expand it to other languages.**

Keywords—Minimal Keystrokes; Productivity; Modification; Similar elements; Syntax tree;

I. Introduction

Programming is a challenging job that often requires typing long codes via a keyboard. Many source code editors and tools such as Visual Studio and Eclipse are developed to help programmers to improve programming productivity. In a source code editor, code completion is a widely used productivity feature. It involves predicting program element such as a word or phrase that the programmer wants to type in without actually typing it in completely, and provides a progressively refined list of candidates matching the input to allow them to choose the right one. This is particularly useful for code writing because it help programmers decrease the number of keystrokes needed to save time spent typing [19]. Moreover, the candidate suggestions can help programmers save time and reduce the errors because often the program8mer will not know exactly what members a particular class has and even the correct spelling of an element. Furthermore, an improved code completion can complete multiple keywords from abbreviated input. One case study about it found a 30% reduction in time usage and a 41% reduction of keystrokes over conventional code completion [2]. Besides the above-mentioned code completion, there are many studies [22, 23] and tools [20, 21] about program element typing －the core task of code writing－ to improve coding efficiency. However, few studies have focused on the operations about selecting texts and moving the cursor. The only widely known fact is that they are supported by keyboard and mouse shortcut keys. The minimum number of keystrokes (hereafter referred to as minimal keystrokes) can be used to determine the fewest number of keystrokes necessary to accomplish a specific typing task [1]. Minimal keystrokes occur when a programmer has a clear goal. Hence, programming productivity should increase as the number of keystrokes is reduced.

Programmers are often faced with programming tasks of many continuous similar operations. For example, 1) ten local variables need to be initialized in a method, 2) an array must be initialized by explicitly setting ten elements, 3) and it is representative that each case block calls a logical method and an output method, with different parameters like List 1, etc. For these specific tasks, some programmers will type all of the code by hand, while others employ the Copy-Paste method [3].

The Copy-Paste method has three steps: 1) Type a representative part of the code. 2) Copy and paste the representative code. 3) Modify the elements as needed to accomplish the task. Like these tasks and operations there are

```
1  switch (intSelector)
2  {
3     case 111: // Pattern 2
4        this.GetMultiply(local_int_1, strNum[intSelector]);
5        Console.WriteLine("The first case."); // Pattern 1
6        break;
7     case 222:
8        this.GetMultiply(local_int_2, strNum[intSelector]);
9        Console.WriteLine("The second case.");
10       break;
11    case 333:
12       this.GetMultiply(local_int_3, strNum[intSelector]);
13       Console.WriteLine("The third case.");
14       break;
15    //.......
16 }
```

List 1. Example of a switch block in C#.

many similar code fragments in the source code.

Similar code is also called code clone or duplicated code and it is one factor that makes software maintenance more difficult [4, 5]. If programmers modify one similar code fragment, then they must determine if the same modification is applicable to other code fragments. Furthermore, similar code fragments sometimes involve similar defects caused by the same mistake [6].

Code clone detection offers effective means to identify similar code, and it is very useful for software analysis, maintenance, and reengineering [7, 8]. Several tools address the problem of identifying code clones such as the ones from the copy-paste modifications [9], and some approaches support programmers in modification tasks that affect different source code locations by automatically eliciting past changes [10]. However, a tool to minimize the number of keystrokes during modifications of similar code does not exist.

To improve programming productivity, we propose an approach to extract the similarity pattern from recently modified elements and provide all possible matching elements as modification suggestions for programmers. Because syntax highlighting helps programmers find errors, the matching elements are highlighted and the next element can be selected by shortcut keys. Finally to improve program productivity, a visual studio extension is implemented.

Specifically this work aims to answer three research questions.

RQ1: Does our tool reduce the minimal keystrokes?

RQ2: Can our tool improve the programming productivity?

RQ3: Does our tool run smoothly without inconvenience?

The contributions of this paper are:

- Proposal of an approach to extract similar elements by analyzing recently modified elements.
- SimilarHighlight, a novel tool that reduces keystrokes by suggesting program elements to modify.
- Demonstration that SimilarHighlight can help improve programming productivity.

SimilarHighlight is released as open source software in https://github.com/youfbi008/SimilarHighlight/ and the tool has been published in Visual Studio Gallery http://goo.gl/KqtTvY.

The remainder of this paper is organized as follows. Section II provides a motivating example. Section III describes our proposed approach and tool, SimilarHighlight. Sections IV and V provide details and evaluate its functions, respectively. Section VI describes related works. Finally, Section VII is the conclusion and future work.

II. Motivating Example

This section provides examples to demonstrate our approach and tool. Consider List 1, which shows a switch block and at least three case blocks, where each case block has a method with two parameters and a system output method.

Generally the Copy-Paste method is used for this programming task. Initially the code typed in the first case is copied and pasted multiple times. Then the elements are modified for each case. In this example, the modified elements are ***case*** keywords, the first parameters of the ***GetMultiply*** methods, and the parameters of the ***Console.WriteLine*** methods. Due to its simplicity, a proficient typist often employs the Copy-Paste method. The more these similar operations, the higher efficiency obtained by the Copy-Paste method.

This study focuses on the third step of the Copy-Paste method, which is similar to modification tasks that often occur in software maintenance and reengineering. An element may be a local variable, a parameter to a method, an expression, a program block consisting of multiple elements, etc. Program elements with similar positions in similar code fragments are defined as similar program elements.

Representative patterns of similar program elements are as follows: (1) Method parameters and (2) Case values of a switch (List 1). In addition, the example in List 2 shows other representative patterns such as (3) Array elements, (4) Local variable names or values, and (5) Method names. To modify similar elements continuously, programmers generally select the whole text of each element and type the new text sequentially. Below, select operations are discussed in detail [11].

A person who usually selects items using a mouse often has two text-selection methods: double-click and click-and-drag. However, the double-click method cannot select the whole parameter text because it just selects a word. Hence, programmers have to click and drag the mouse over the whole text to accomplish this operation.

A person who usually selects items using a keyboard, especially the shortcut keys, often has two text-selection methods: [Shift]+arrow and [Ctrl]+[Shift]+[Right arrow] | [Left arrow]. The latter method can select from the current position to the right or left of the current word. Thus, to select the whole text such as ***"The first case."***, the arrow key must be pressed four times.

```
1   void function_A(int a, int b)  // Pattern 5
2   {
3      string[] strNum = new string[] {
4        "one", "two", "three", "four", "five", "six", "seven",
5   "eight", "nine",
6      }; // Pattern 3
7   }
8
9   void function_B()  // Pattern 5
10  {
11     int local_int_C = 111; // Pattern 4
12     string local_String_D = "Hello world";
13  }
```

List 2. Example of modification patterns in C#.

Using both a mouse and keyword effectively should be more convenient. However, some appropriate subjects should be considered when many similar program elements must be modified, especially if the elements are scattered throughout the source file. Identifying every necessary modification is time-consuming and often modifications are missed. Additionally, selecting the text of each element is a hassle in a continuous modification.

To illustrate these issues, we conducted an experiment involving a person who uses a keyboard where a programming task is composed of patterns where each pattern has nine similar elements. The text of the similar elements should be continuously rewritten. To present the proportion of the keystrokes to select texts and move the cursor in the entire task, each keystroke is counted separately to determine the minimal keystrokes. Figure 1 shows the percentages of keystrokes for selecting and moving operations.

At least 33% of the minimal keystrokes are used to select texts and move the cursor, but this value can be as high as 60% for shorter text (Fig. 1). Additionally, when elements are further separated in the code, more keystrokes are used to move between elements, resulting in more unnecessary keystrokes. Thus, the cost of the keystrokes for selecting and moving operations should not be neglected in programming. Consequently, reducing the number of keystrokes should improve programming productivity.

III. SimilarHighlight: A Tool to Improve Programming Productivity

We propose a tool (SimilarHighlight) to help programmers improve their productivity. Our tool suggests program elements similar to the last selected element that might be modified during the next modifications. The elements are highlighted and the text of the next element can be selected immediately by shortcut keys for easy modification. Figure 2 summarizes the main steps of SimilarHighlight.

First, the source code file is parsed into a concrete syntax tree (CST) [12] similar to the XML DOM by the Code2Xml library [13]. A program element can be represented as a single node or a subtree. Two different elements of the last selected elements are compared to extract the common node set as a similarity pattern. In addition, candidate node type is extracted

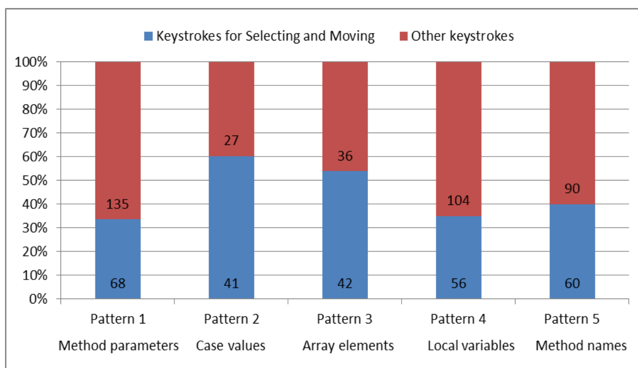

Fig. 1. Minimal keystrokes comparison without our tool.

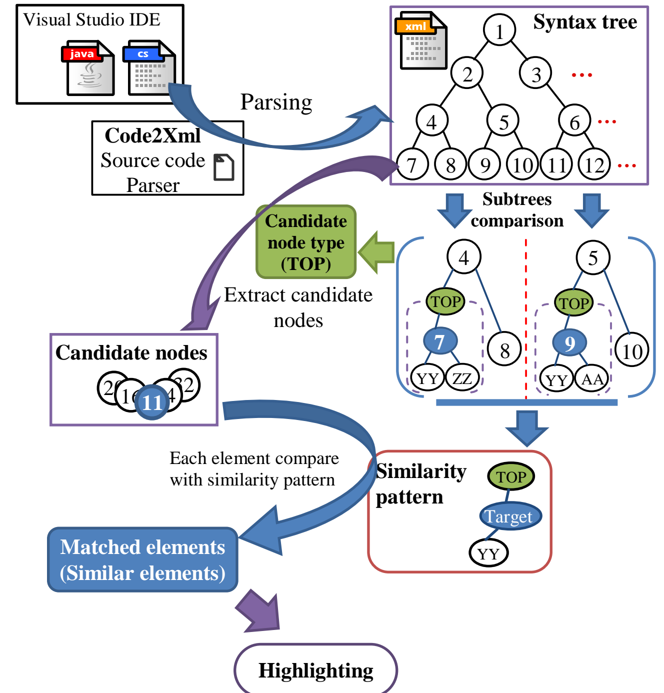

Fig. 2. Overview of SimilarHighlight.

to determine the candidate nodes. Second, each of the candidate nodes are compared to the similarity pattern to check whether they match. Finally, SimilarHighlight highlights all the corresponding elements of the matching nodes and presents them to the programmers.

A. *Parsing a source file into a concrete syntax tree and determining the corresponding subtree of an element*

The source code of a source file is called a compilation unit in C#, JAVA, etc. A compilation unit normally contains a single class definition that is parsed into a CST by the Code2Xml library. Code2Xml is a set of parsers to interconvert between the source code and xml supporting multiple programming languages. Due to Code2Xml, SimilarHighlight supports C, C#, JAVA, JavaScript, and PHP, and should support other languages in the future.

A program element is usually represented as a single node in the syntax tree. However, a program element in our CST is represented as a subtree, which consists of multiple nodes, including a token node. Each node has node type and node id. If the node is a token node, it also has positional data to describe the position of the element in the source code.

As an easy-to-understand example, List 3 shows a parsed xml where the xml texts for ***Console.WriteLine("The first case.")*** is omitted. Although the complete xml text is ten times longer, the main elements such as *(*, ***"The first case."***, and *)* in this expression are presented in red. In our approach, selecting the element of ***"The first case."*** via a mouse or keyboard causes SimilarHighlight to determine the corresponding token node by comparing the cursor positional data and the node positional data, such as *startline*, *startpos*, *endline*, and *endpos*. To represent the corresponding subtree of the current element,

```
1   <brackets_or_arguments id="257">
2    <arguments id="276">
3     <TOKENS id="char_literal279">
4      <TOKEN id="char_literal279" …>(</TOKEN>
5     </TOKENS>
6     <argument_list id="280">
7      <STRINGLITERAL id="set1275">
8       <TOKEN id="set1275" startline="86" startpos="38"
9   endline="86" endpos="55">"The first case."</TOKEN>
10     </STRINGLITERAL>
11    </argument_list>
12    <RPAREN id="char_literal281">
13     <TOKEN id="char_literal281" …>)</TOKEN>
14    </RPAREN>
15   </arguments>
16  </brackets_or_arguments>
```

List 3. Omitted xml text of the syntax tree about the expression: ***Console.WriteLine("The first case.");***.

outermost ancestor also must be determined. The outermost ancestor is outermost one in the ancestors which is ancestor of the token node and has no other immediate child nodes. Then the corresponding subtree can be represented by the nodes from the outermost ancestor to the current node. Figure 3 shows the corresponding subtree for ***"The first case."*** and the node types used to describe the nodes. In addition, the outermost ancestor type, which is seen as the type of element in CST, is used to extract the candidates. The outermost node type of ***"The first case."*** is ***argument_list***.

Because an expression can be seen as an element in our approach, Fig. 4 shows the corresponding subtree of the expression: ***Console.WriteLine("The first case.");*** as an element in our approach. Although some nodes are omitted, the structure and position can be understood. The next step considers the subtree to determine the surrounding nodes of ***"The first case."***.

B. Extracting the similarity pattern

In the example of List 1, when the parameter texts of ***Console.WriteLine*** in the first two case blocks: ***"The first case."*** and ***"The second case."*** are selected successively, the corresponding subtrees of the two elements can be determined as mentioned above. Then SimilarHighlight will compare their surrounding nodes. The surrounding nodes generally consist of ancestor nodes, sibling nodes, and descendant nodes. Because

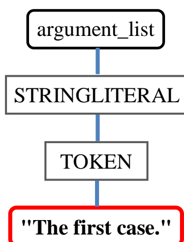

Fig. 3. Corresponding subtree of the element: ***"The first case."***.

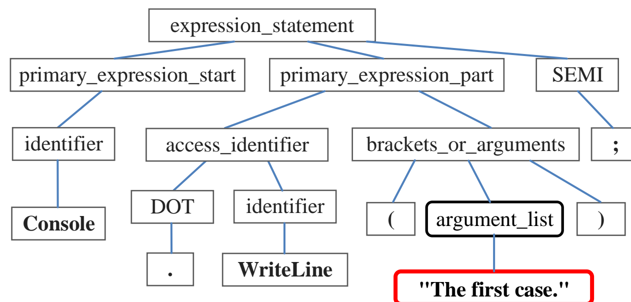

Fig. 4. Omitted subtree of the expression: ***Console.WriteLine("The first case.");***.

it is important to effectively collect this information, our approach extracts the surrounding nodes from CST. This information is then used to construct a node set. List 4 shows a case with an omitted node set for ***"The first case."***. The numbers on the left are the index of the data in the node set. The non-consecutive index numbers indicate that too much data is omitted to understand the relationships between List 3 and 4. In practice, the node type, node id, and token text shown in List 3 is used to construct the data of the node set in List 4. The main elements of the expression such as ***Console***, ***WriteLine*** marked in red can be found in the node set, hence the main elements are seen as the surrounding nodes as we expected.

List 5 shows a pseudocode to present our approach in collecting the surrounding nodes. First, the traversal from the outermost node to the token node is presented as an index of 0 to 22. Next, two methods are used to determine other surrounding nodes: (1) find the immediate child nodes of all new added nodes and (2) find sibling nodes of the immediate parent node. To collect more accurate data, these methods are repeated several times.

Figure 5 compares the node sets to extract the similarity pattern. The data of the two node sets are similar, excluding

```
[0]:<argument_list280
[1]:<argument_list280<argument185
[2]:<argument_list280<argument185<argument_value190
[3]:<argument_list280<argument185<argument_value190<expression193
......
[22]:<argument_list280<argument185<argument_value190<expression193<......<primar
y_expression_start232<literal242<STRINGLITERALset1275
[23]:argument_list280
[24]:argument_list>'"The first case."'
[25]:argument_list-'('
[26]:argument_list-')'
......
[28]:argument_list-TOKENSchar_literal279(
[29]:argument_list-RPARENchar_literal281)
......
[40]:argument_list<arguments276<brackets_or_arguments257-access_identifier256>'.'
[41]:argument_list<arguments276<brackets_or_arguments257-
access_identifier256>'WriteLine'
[42]:argument_list<arguments276<brackets_or_arguments257<primary_expression_part
233-'Console'
......
[56]:argument_list<arguments276<brackets_or_arguments257<primary_expression_part2
33<primary_expression210<primary_or_array_creation_expression163
```

List 4. Omitted node set of the surrounding nodes of the element: ***"The first case."***.

```
Add outermost ancestor to the Child node set
SET outermost ancestor to the Parent node

FOR each node from outermost ancestor to immediate parent of token
    Add the node to the Result node set
ENDFOR

Add outermost ancestor to the Result node set

FOR loop one to many times
    FOR each node in the Child node set
        FOR each child node of the node
            Add the child node to the new Child node set
            Add the child node to the Result node set
        ENDFOR
    ENDFOR

    FOR each node in the first ten siblings of the Parent node
        Add the node to the new Child node set
        Add the node to the Result node set
    ENDFOR

    SET the new Child node set to the Child node set
    SET parent of the Parent node to the Parent node
ENDFOR
```

List 5. The pseudo code for collecting surrounding nodes.

index 24 and other omitted data. In practice, there are 52 common data points. Therefore, the elements are similar because they have many similar surrounding nodes. Then the common data of node sets are defined as the similarity pattern.

C. Extracting all possible matching elements

To ensure a high running performance, each program element in the source file cannot be traversed to verify similar elements. Fortunately, candidates can be extracted using the outermost node type of CST (as mentioned in 3.1). Figure 6 shows the process to determine similar elements. Because the outermost node types of the two elements are both ***argument_list***, the outermost node type is a candidate node type. Then all nodes where the outermost node type is ***argument_list*** are extracted as candidate nodes, and the surrounding nodes of each candidate node are compared to the similarity pattern. A preset threshold is used to determine if there is enough common data to be a valid match (i.e., the corresponding element of the node is a similar element). Finally, similar elements are highlighted based on the positional data of the corresponding token nodes.

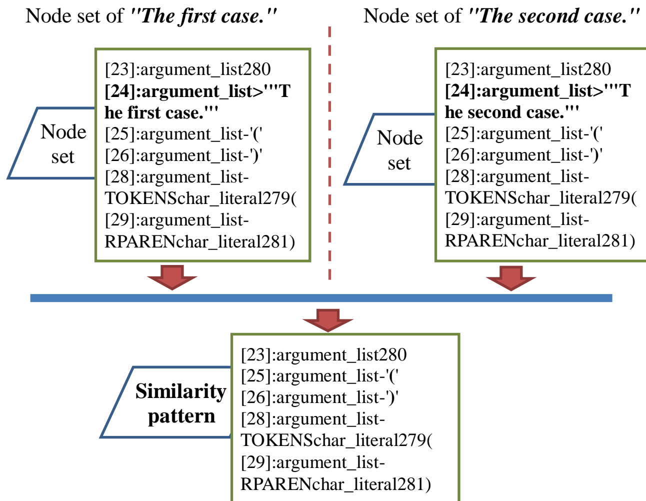

Fig. 5. Comparison of node sets to extract similarity pattern.

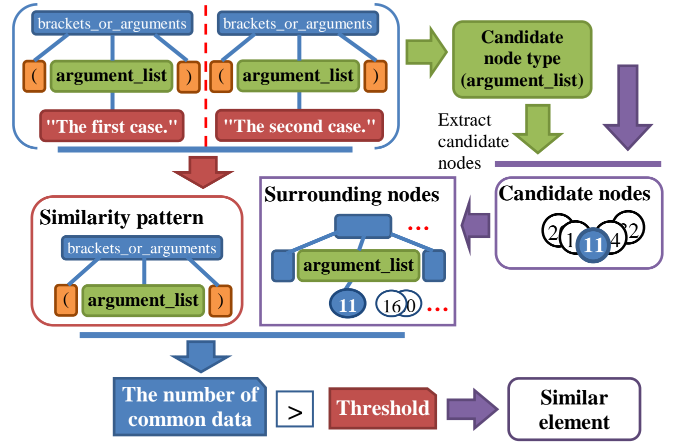

Fig. 6. Comparison of node sets to extract the similarity pattern.

IV. VISUAL STUDIO EXTENSION

SimilarHighlight is implemented in a visual studio extension to evaluate our approach and to help programmers improve their programming productivity. The main functions of the SimilarHighlight are as follows:

- Elements similar to the last selected element are highlighted.
- The previous or next similar element can be found immediately via shortcut keys, and the whole text is selected for easy modification.
- A margin is added on the right side in the visual studio editor to offer relative position markers of similar elements.
- A pane named “Similar” is added into the output window to provide more information about similar elements.
- Some settings in the tool can be customized, including enable (disable) the functions and adjusting the similarity level to change the threshold to improve or reduce the scope of similar elements.

To present the functions of the tool, Fig. 7 shows the running results of a more complicated example than the motivating example. In the parameter texts of ***Console.WriteLine*** in the first two case blocks: ***first*** and ***second*** are selected successively using a mouse or keyboard, and similar elements are obviously highlighted. Although the whole text in the token has double quotations like ***"The first case."*** not ***The first case***, for quick modifications the double quotations are ignored. The current cursor is located in the second case block whose background color is deeper than others, but the next similar element can be found by ***Ctrl + Alt***

```
17      case 1:
18          this.GetMultiply(local_int_1, strNum[intSelector]);
19          Console.WriteLine("The first case.");
20          break;
21      case 2:
22          this.GetMultiply(local_int_2, strNum[intSelector]);
23          Console.WriteLine("The second case.");
24          break;
25      case 3:
26          this.GetMultiply(local_int_3, strNum[intSelector]);
27          Console.WriteLine("The third case.");
28          break;
29      case 4:
30          Console.WriteLine(   "fourth"   );
31          break;
32      case 5:
33          Console.WriteLine("case:" + intSelector);
34          break;
```

```
Output
Show output from: Similar
Selection: 55, Line: 19, Range: (44, 49), Code: first
Selection: 56, Line: 23, Range: (44, 50), Code: second
================================Start:19==============================
The max similarity is 52.
Line: 24, Range: (39, 57), Similarity: 52, Code: "The second case."
Line: 28, Range: (39, 56), Similarity: 52, Code: "The third case."
Line: 31, Range: (41, 49), Similarity: 52, Code: "fourth"
Line: 20, Range: (39, 56), Similarity: 52, Code: "The first case."
Line: 34, Range: (39, 60), Similarity: 45, Code: "case:" + intSelector
=================================End:19===============================
```

Fig. 7. Running result of SimilarHighlight.

+ ***Right Arrow.*** Then the text of the next element can be modified immediately. Consequently, many select and move operations become unnecessary, reducing the minimal keystrokes.

Furthermore, another technique worth mentioning is to click a mark using the left mouse button in the right margin to select the corresponding element of that mark. This allows an quick jump to another type of element. An additional function is that the "**Similar**" output window is used to offer selected element information, which provides text and similarity in order. The similarity is a count of common data in the similarity pattern. In this example, the maximum similarity is 52. The similarity of the element in fifth case block is 45, which exceeds the predetermined threshold. In addition, although part of the text for the element is selected by the mouse or keyboard, the element can be found exactly if the source code in the file does not have a serious format error.

V. EVALUATION

To assess the effectiveness of SimilarHighlight, we conducted a set of experiments and compared the results against conventional methods to answer the three research questions.

A. Experiment 1

To investigate RQ1 (Does our tool reduce the minimal keystrokes?), we reevaluated the experiment in the Motivating Example using our tool. Then the minimal keystrokes for the selecting and moving operations with and without using our tool were compared to calculate the reduction rates.

Figure 8 compares the minimal keystrokes for the five

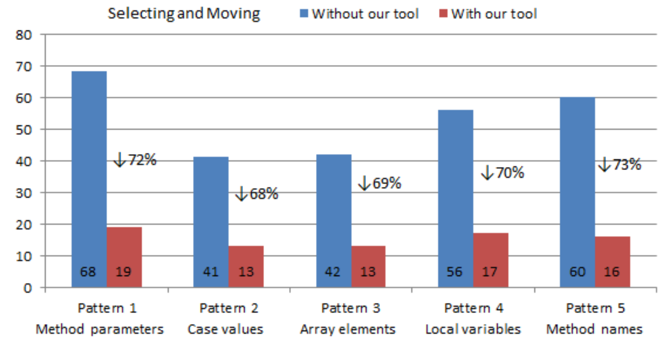

Fig. 8. Minimal keystrokes comparison.

similar element patterns and the reduction rates. Our tool results in an almost 70% reduction in the minimal keystrokes for selecting texts and moving the cursor. In particular, the longer the distance between each element, the higher productivity.

Therefore, SimilarHighlight can significantly reduce the minimal keystrokes for selecting and moving in a modification task.

B. Experiment 2

To investigate RQ2 (Can our tool improve the programming productivity?), we conducted an experiment consisting of two modification tasks for a person using a keyboard. The first one contains an array of 20 elements similar to pattern 3. The second one is more complex and it consists of ten case blocks in a switch block similar to Fig. 1, which includes pattern 1 and pattern 2 (https://github.com/youfbi008/SimilarHighlight/blob/master/SimilarHighlight.Tests/SimilarityTest1.cs) This experiment tests the third step of the Copy-Paste method (element modification). We measured the time and the keystrokes necessary to accomplish each task with and without our tool. The results were compared to calculate the reduction rates. It will not have a beneficial effect if we use our tool base on without tool using in the experiment, because the methods of operations are different. Eight master's degree students studying computer science (S1, S2 …, and S8) participated in the experiments.

Figure 9 and 10 show the results for the first and second

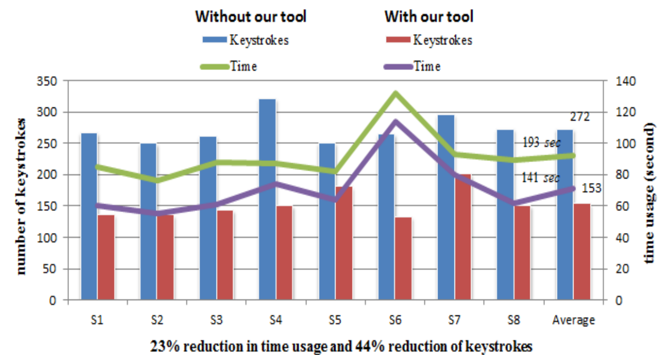

Fig. 9. Running results in the first task.

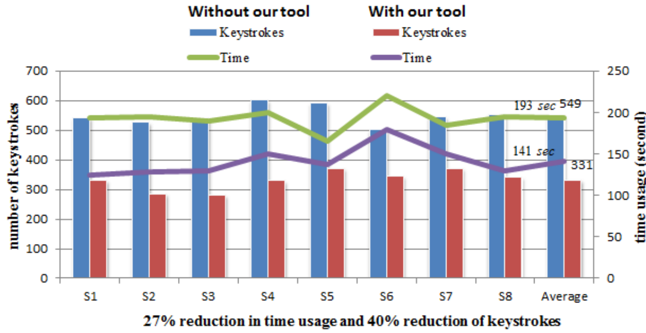

Fig. 10. Running results in the second task.

tasks, respectively. The averages of time usage and the number of keystrokes were calculated to determine the reduction rate in using our tool. SimilarHighlight reduces the coding time by approximately 23% and the keystrokes by 44% in the first task (Fig. 9). Because the reduction in keystrokes is nearly twice the reduction in time, as programmers become more familiar with our tool, the time reduction should become larger.

SimilarHighlight reduces the coding time by approximately 27% and the keystrokes by 40% in the second task (Fig. 10). Similarly, familiarity with the tool is important to further reduce the coding time.

These experiments demonstrate that SimilarHighlight can reduce costs of writing code and improve programming productivity, especially when a keyboard is used. The difference in the time usage and the number of keystrokes between our method and conventional methods was statistically significant based on *wilcoxon signed-rank tests* (p-value < 0.05) [25]. Therefore, our method is significantly better than conventional methods.

C. Experiment 3

Because the parsed xml text becomes too long as the line number of the source code file increases and often there are too many candidate elements, the running performance of our tool is considered. It is possible that our tool does not run smoothly or is inconvenient to programmers. To investigate RQ3 (Does our tool run smoothly without inconvenience?), we conducted an experiment in which our tool was used to determine similar elements in five files (https://github.com/youfbi008/SimilarHighlight). The number of similar elements (CNT) and the average running time (ART) was recorded separately.

Figure 11 shows the running results. The test file names and the source lines of code (SLOC) are listed on the top and bottom separately in the order of increasing SLOC. Then ART is presented as the bar and CNT is presented as the number over the top of the corresponding bar. As the running results, source files with less than 5000 SLOC ran in less than 1 second. Because the elements are highlighted earlier using our tool rather than the default highlighting functionality of visual studio [14], programmers do not have to wait for elements to be highlighted to continue with the next operation. Additionally, the main process steps of SimilarHighlight run in background thread, which minimizes the wait time. Therefore, SimilarHighlight runs smoothly without affecting the operations.

VI. Related Works

There are many research works to detect similar code, especially about clone detection techniques. Four main approaches, namely string-based, token-based, tree-based and PDG-based, are used by source code similarity detection tools.

Ducasse et al. [16] proposed a language independent approach which is String-based. The approach works on the source code directly to look for specific patterns in a comparison from every line to every other.

Kamiya et al. [5] provide a token-based code clone detection tool named CCFinder, which transforms tokens of a program according to a language-specific rule and performs a token-by-token comparison.

Because the parse tree (CST) and abstract syntax tree (AST) contains the complete information of the source code, the matches of subtrees can be identified by comparing subtrees within the generated tree [15]. Our approach is also tree-based. However, because of the different aim, we use the subtree comparison to find out the similar elements.

PDG is program dependence graph which is a representation of a program that represents only the control and data dependency among statements [17]. Krinke et al. [24] uses

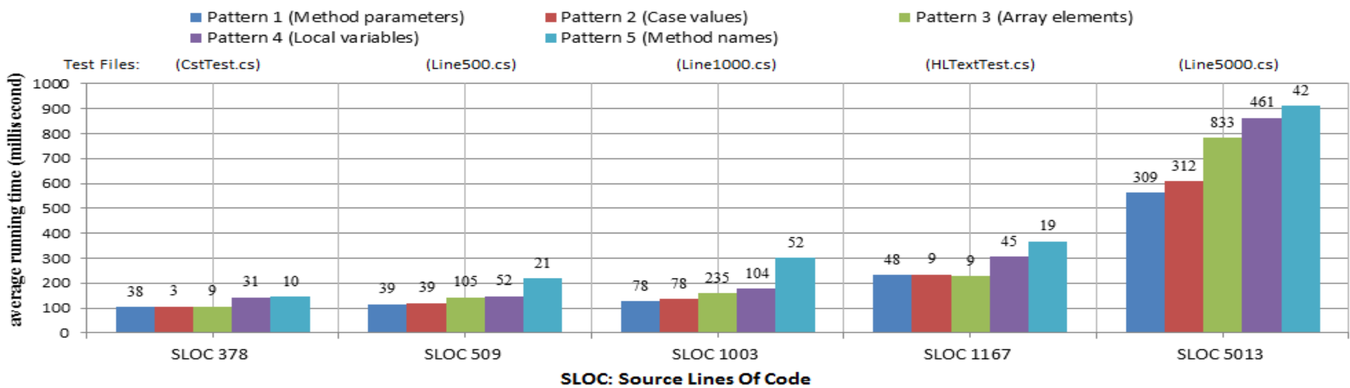

Fig. 11. Running results of the five patterns.

the PDG-based method to detect maximal similar subgraphs.

Due to the different aim, our approach is to find similar code in one source file, not in the entire project. We find the similar program elements not the code fragments. Our tool, SimilarHighlight, suggests the programmer to modify them at the next modifications and reduce the keystrokes to improve the programming productivity.

VII. Conclusions and Future Works

We elucidated problems in successive modifications through motivating examples and developed a tool called SimilarHighlight to resolve the problems. SimilarHighlight suggests program elements similar to the last selected elements that could be modified during the next modification. These suggested elements are highlighted and their text can be selected immediately by shortcut keys, reducing the minimal keystrokes. Moreover, we evaluated the effectiveness of SimilarHighlight in empirical experiments.

Our tool can be used in programming tasks and modification tasks to improve the programming productivity. Furthermore, source code review is a peer review of the source code of computer programs. It is intended to find and fix defects overlooked in early development phases, improving overall code quality [18]. Additionally, highlighting similar elements can easily identify elements, especially when reviewing for consistency.

Our aim is to make SimilarHighlight the default functionality of the source code editor. In the future, we will improve our approach and our tool as follows:

- Improve the running performance. Although the average running time is less than 1 second, it can be improved, especially when the SLOC exceeds 3000.
- Improve the precision to match similar elements, which may encourage more programmers to use our tool.
- Support more programming languages. Currently SimilarHighlight can be used in C, C#, JAVA, JavaScript, and PHP files. We are contributing to a Code2Xml project to support more programming languages, such as Cobol.
- Extract more patterns based on programming habits. Although programming habits vary by programmer, we intend to extract potential modification patterns. Additionally, instead of highlighting all of the text of an element, we will highlight only the part to be modified.
- Add a suggestion list about text modifications similar to Code Completion. When the next element is selected by shortcut keys, a list of modification suggestions will be displayed based on the modification history of similar elements.

References

[1] H. Duan and B. P. Hsu, “Online spelling correction for query completion,” in Proc. WWW. New York, USA, 2011, pp. 117–126.

[2] S. Han, D. R. Wallace, and R. C. Miller, “Code completion from abbreviated input,” in Proc. ASE. IEEE Computer Society, 2009, pp. 332–343.

[3] M. Kim, L. D. Bergman, T. A. Lau, and D. Notkin, “An ethnographic study of copy and paste programming practices in OOPL,” in Proc. ISESE, 2004, pp. 83–92.

[4] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, “Clone detection using abstract syntax trees,” in Proc. ICSM, 1998, pp. 368–377.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Trans. Softw. Eng., vol. 28, no. 7, pp. 654–670, 2002.

[6] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl, “Assessing the benefits of incorporating function clone detection in a development process,” in Proc. ICSM, Oct. 1997, pp 314–321.

[7] M. Gabel, L. Jiang, and Z. Su, “Scalable detection of semantic clones,” in Proc. ICSE, New York, USA, 2008, pp. 321-330.

[8] E. Burd and J. Bailey, “Evaluating clone detection tools for use during preventative maintenance,” in Proc. SCAM, Montreal, Canada, Oct. 2002, pp. 36-43.

[9] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and Evaluation of Clone Detection Tools,” IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 577-591, Sep. 2007.

[10] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, “Predicting source code changes by mining change history,” IEEE Trans. Softw. Eng., vol. 30, no. 9, pp. 574-586, 2004.

[11] 15 ways to select text in a Word document, http://www.techrepublic.com/blog/microsoft-office/15-ways-to-select-text-in-a-word-document/

[12] Parse tree, http://en.wikipedia.org/wiki/Parse_tree

[13] Code2Xml, https://github.com/exKAZUu/Code2Xml

[14] Microsoft: How to: Use Reference Highlighting, http://msdn.microsoft.com/en-us/library/vstudio/ee349251(v=vs.100).aspx

[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. “Clone detection using abstract syntax trees,” in Proc. ICSM, 1998, pp. 368–377.

[16] S. Ducasse, M. Rieger, and S. Demeyer. “A Language Independent Approach for Detecting Duplicated Code,” in Proc. IEEE Int’l Conf. on Software Maintenance (ICSM), Oxford, England, Aug. 1999, pp. 109-118.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 9, no.3, pp. 319-349, 1987.

[18] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto, “Analyzing individual performance of source code review using reviewers' eye movement,” in Proc. Eye tracking research & applications (ETRA), San Diego, California, 2006, pp. 133-140.

[19] D. Anson, P. Moist, M. Przywars, H. Wells, H. Saylor, and H. Maxime. “The effects of word completion and word prediction on typing rates using on-screen keyboards,” Assistive technology, vol. 18, no. 2, pp. 146-154, 2006.

[20] IntelliJ IDEA, http://www.jetbrains.com/idea/

[21] Resharper, http://www.jetbrains.com/resharper/

[22] G. Little and R. C. Miller, “Keyword Programming in Java,” Autom. Softw. Eng., vol. 16, no. 1, pp. 37-71, 2009.

[23] K. Czarnecki and U. Eisenecker. Generative Programming - Methods, Tools, and Applications. Boston : Addison Wesley, 2000.

[24] J. Krinke, “Identifying Similar Code with Program Dependence Graphs,” in Proc. Working Conf. Reverse Eng., 2001, pp. 301-309.

[25] Wilcoxon signed-rank test , http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test