A Tool for Suggesting Similar Program Element Modifications

YUJIANG YANG[†1] KAZUNORI SAKAMOTO[†2]
HIRONORI WASHIZAKI[†1] YOSHIAKI FUKAZAWA[†1]

Many programming tasks require programmers to modify similar program elements continuously. It will take some time to find out the next element to be modified without missing the necessary modifications, and it is too much hassle to select the text to change each similar element. To improve these problems, we extracted all possible matched elements by using the similarity patterns from recently modified elements. The sub syntax tree comparison is employed to extract similarity patterns. As a tool, SimilarHighlight can give programmers some suggests that program elements are similar to the last selected elements and might be modified at the next modifications. The elements will be highlighted and the text of the next element can be selected immediately for modify by shortcut keys. In addition, the tool supports C#, Java, C, JavaScript and other languages in future.

1. Introduction

As we know, programming is a challenging job that often requires programmers to write a lot of code by keyboard typing. As the minimal keystrokes are used to measure the minimal number of key presses the user has to make in order to accomplish a specific typing task [1], a programming task has minimal keystrokes when programmer has a clear goal. A user study found 30% reduction in time usage and 41% reduction of keystrokes over conventional code completion [2]. The programming effort will be reduced if we decrease the minimal keystrokes, therefore, the programming productivity should increase.

Programmers are often faced with programming tasks that they have to do many similar operations continuously. For example, ten local variables need to be initialized in the method, or an array must be initialized by explicitly setting ten elements, which is more representative when in a switch block that each case block call a logical method and an output method, but the parameters are different like List 1, etc. For these specific tasks, some programmers will type all of the code by hand, but the others maybe accomplish these tasks by using the Copy-Paste method [3] like the following: 1) Type a representative part of all code. 2) Copy the part code, and paste them to reach the amount of code. 3) Modify the elements as expected to accomplish the task.

Similar code is generally considered as one of factors that make software maintenance more difficult [4, 5]. If developers modify one of similar code fragments, they have to determine whether or not to apply the same modification to the others. Furthermore, similar code fragments sometimes involve similar defects caused by the same mistake [6].

Similar code is also called ***code clone***. Software analysis, maintenance and reengineering could often benefit from performing clone detection [7, 8]. Several tools address the problem of identifying software clones that come from copy-paste modifications [9] and some approaches support developers in modification tasks that affect different source code locations by automatically eliciting past changes [10]. However, there is no tool to help developers reduce their minimal keystrokes in the modifications of similar code.

To increase the programming productivity, we propose an approach to extract the similarity pattern from recently modified elements, and to extract all possible matched elements as modification suggestions for programmers. As syntax highlighting also helps programmers find errors in their program. The matched elements are highlighted and the next element can be selected by shortcut keys. Finally, a visual studio extension is developed to evaluate the approach and implement it to help programmers increase their programming productivity.

```
switch (intSelector)
{
    case 111: // Pattern 2
        this.GetMultiply(local_int_1, strNum[intSelector]);
        Console.WriteLine("The first case."); // Pattern 1
        break;
    case 222:
        this.GetMultiply(local_int_2, strNum[intSelector]);
        Console.WriteLine("The second case.");
        break;
    case 333:
        this.GetMultiply(local_int_3, strNum[intSelector]);
        Console.WriteLine("The third case.");
        break;
    .......
}
```

List 1. An example of the switch block in C#

The contributions of this paper are as follows:

- Proposal of an approach to extract the similar elements by analyzing recently modified elements.
- SimilarHighlight, a tool for suggesting program elements might be modified at the next modifications.
- An evaluation of SimilarHighlight to show that our tool can help programmers increase their programming productivity.

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

The remainder of this paper is organized as follows. First, we provide a motivating example to explain our work in Section 2. In Section 3, we describe our proposed approach and tool, Sim-

†1 Waseda University
†2 National Institute of Informatics

ilarHighlight. And the functions of the tool are described more detail in Section 4. Then we discuss our evaluation in Section 5 and discuss related work in Section 6. Finally, we provide a conclusion and future works in Section 7.

2. Motivating example

In this section, some examples are taken to demonstrate our approach and tool. The example of List 1 is very appropriate to demonstrate them in practice, which shows a switch block and at least three case blocks. And each case block consist of two methods: a member method with two parameters and a system output method.

The Copy-Paste method as mentioned above are generally used to accomplish this programming task. That is need to type the code of the first case, and copy the first case and pastes it multiple times. Then, the elements just need to be modified, which should be modified namely the numbers behind of the ***case*** keywords, the first parameters of the ***GetMultiply*** methods, and the parameters of the ***Console.WriteLine*** methods. They are not complicated for programmers, as good keyboard operators to use the Copy-Paste method. The more similar operations will be done, the higher efficiency will be obtained by the Copy-Paste method.

In this study, the third step of the Copy-Paste method will be paid attention to, and it is similar to some modification tasks that often occur in software maintenance and reengineering in practice. It is generally known that the element could be a local variable, a parameter to a method and even an expression or a program block consisting of multiple elements, etc. The program elements have similar positions in similar code fragments are defined as similar program elements.

The representative patterns of similar program elements are as follows: 1) parameters of methods and 2) case values of a switch, which can be found in List 1. In addition, the example in List 2 shows the other representative patterns such as 3) array elements, 4) local variable names or values, 5) method names. To modify the similar elements continuously, programmers generally have to select the whole text of each element and type the new text sequentially. Next, the select operations will be discussed in detail [11].

```
void function_A(int a, int b)
{
        string[] strNum = new string[] {
         "one", "two", "three", "four", "five", "six", "seven", "eight",
"nine",
        }; // Pattern 3
}

void function_B() // Pattern 5
{
        int local_int_C = 111; // Pattern 4
        string local_String_D = "Hello world";
}
```

List 2. An example of modification patterns in C#

A mouse person who usually selects something by using mouse in programming often has two methods to select text: double-click and click-and-drag. However, the double-click method can't select the whole parameter text, because it just can select a word. So programmers have to click and drag the mouse over the whole text to accomplish the select operation.

A keyboard person who usually selects something by using keyboard especially the shortcut keys often has two methods to select text too: [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, so it almost just need the arrow key to be pressed 3 times than the former to select the whole text such as ***"The first case."*** text.

It will be convenient by using the mouse and keyboard effectively. However, some appropriate subjects should be debated namely when the similar program elements need to be modified are very many, maybe they are scattered in the source file. It will take some time to find out the next element needs to be modified, and prevent missing the modifications necessary. Furthermore, it is too much hassle to select the text of each element need to be modified continuously.

We conducted an experiment about keyboard person to describe these problems. Nine similar elements in each pattern are expected to rewrite the text continuously. To present the proportion of the keystrokes for selecting texts and moving in the entire task, the minimal keystrokes of them are counted separately and the percentage are showed in Table 1.

Pattern	**1**	**2**	**3**	**4**	**5**
All keystrokes	203	68	78	160	150
Selecting and moving	68	41	42	56	60
Percentage	33%	60%	54%	35%	40%

Table 1. The minimal keystrokes comparison of non-tool-using

As the table shows, the minimal keystrokes for selecting texts and moving are at least 33% of the all keystrokes, while it is 60% when the each text to be changed is shorter. Furthermore, when the distance between each element is longer, the keystrokes for moving will be in increased. So the cost of the keystrokes for selecting texts and moving are should not be neglected in programming. To increase the programming productivity, the keystrokes are expected to be reduced.

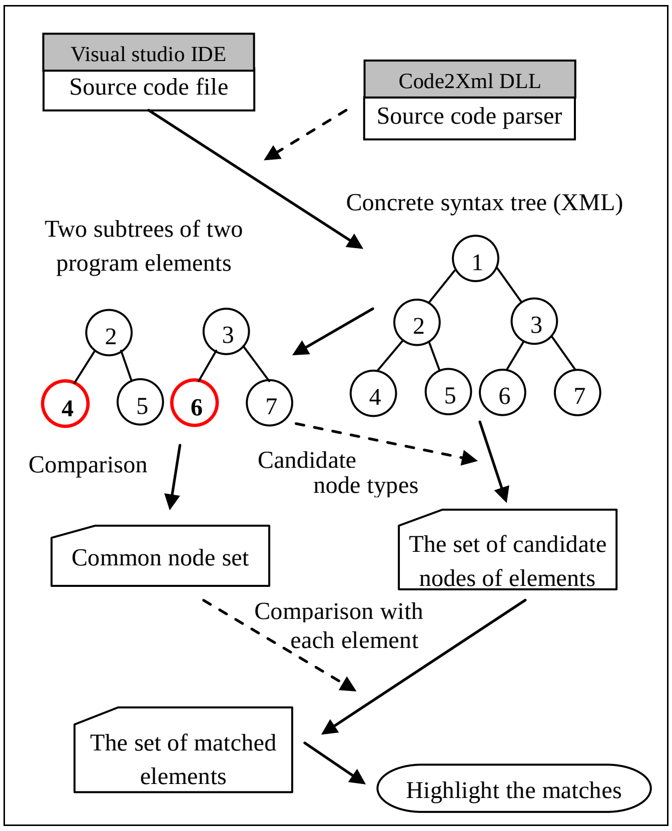
Pattern	**1**	**2**	**3**	**4**	**5**
All keystrokes (tool-using)	154	40	49	121	106
Selecting and moving (tool-using)	19	13	13	17	16
Selecting and moving (non-tool)	68	41	42	56	60
Percentage of non-tool-using	28%	32%	31%	30%	27%

Table 2. The minimal keystrokes comparison of tool-using

Table 2 shows the minimal keystrokes comparison by using SimilarHighlight and the percentage of non-tool-using. Almost 70% of the minimal keystrokes for selecting texts and moving are reduced by using our tool. SimilarHighlight will extract all similar elements of first two modified elements and highlight them. The cursor can be move to the next similar element by shortcut keys immediately, and the whole text of it will be selected to modify easily. Therefore, the longer distance between each element is, the higher productivity will be obtained.

3. SimilarHighlight: a tool to increase programming productivity

An approach is proposed to help programmers increase their programming productivity by SimilarHighlight. It can give programmers some suggests that program elements are similar to the last selected elements and might be modified at the next modifications. The elements will be highlighted and the text of the next element can be selected immediately for easy modify it by shortcut keys.


Visual studio IDE
Source code file
Code2Xml DLL
Source code parser
Concrete syntax tree (XML)
Two subtrees of two program elements
1
2
3
4
5
6
7
Comparison
Candidate node types
Common node set
The set of candidate nodes of elements
Comparison with each element
The set of matched elements
Highlight the matches
Fig. 1. Overview of SimilarHighlight

The main steps of SimilarHighlight are summarized in Figure 1. A source code file is first parsed into a concrete syntax tree (CST) [12] similar to the XML DOM by Code2Xml library [13]. A program element can be represented as a single node or a subtree in this tree. And two different elements of last selected elements will be compared to extract the common node set as the similarity pattern. In addition, candidate node types will be extracted to find out the candidate nodes corresponding to elements. Next, each of the candidate nodes will be compared with the similarity pattern to check whether it is matched. Finally, SimilarHighlight will highlight the all matched elements to present them to programmers.

3.1 Parsing a source file to concrete syntax tree

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

```
<statement id="1096">
  <expression_statement id="1063">
    <primary_expression id="210">
      <primary_expression_start id="232">
        <identifier id="241">
          <IDENTIFIER id="set1277">
            <TOKEN id="set1277" startline="86" startpos="20" endline="86"
endpos="27">Console</TOKEN>
          </IDENTIFIER>
        </identifier>
      </primary_expression_start>
      <primary_expression_part id="233">
        <access_identifier id="256">
          <access_operator id="268">
            <DOT id="set278">
              <TOKEN id="set278" …>.</TOKEN>
            </DOT>
          </access_operator>
          <identifier id="269">
            <IDENTIFIER id="set1277">
              <TOKEN id="set1277" startline="86" startpos="28" end-
line="86" endpos="37">WriteLine</TOKEN>
            </IDENTIFIER>
          </identifier>
        </access_identifier>
        <brackets_or_arguments id="257">
          <arguments id="276">
            <TOKENS id="char_literal279">
              <TOKEN id="char_literal279" …>(</TOKEN>
            </TOKENS>
            <argument_list id="280">
              <STRINGLITERAL id="set1275">
                <TOKEN id="set1275" startline="86" startpos="38" end-
line="86" endpos="55">"The first case."</TOKEN>
              </STRINGLITERAL>
            </argument_list>
            <RPAREN id="char_literal281">
              <TOKEN id="char_literal281" …>)</TOKEN>
            </RPAREN>
          </arguments>
        </brackets_or_arguments>
      </primary_expression_part>
    </primary_expression>
    <SEMI id="char_literal1133">
      <TOKEN id="char_literal1133" …>;</TOKEN>
    </SEMI>
  </expression_statement>
</statement>
```

List 3. Omitted xml text of syntax tree represent an example:
Console.WriteLine("The first case.");

Each program element has node type and position information. To present the parsed xml is easy to understand, that an omitted xml texts of subtree of the example: ***Console.WriteLine("The first case.");***, which is presented in List 3. The complete xml text is triple of it in practice. The main elements in the expression are presented in bold. In this approach, the effective node type to extract the candidates is the node type of the outmost node which is ancestor node of a node and hasn't other direct children node. It can be found that the effective node type of the ***Console*** element is ***primary_expression_start***, while the effective node type of ***"The first case."*** is ***argument_list***.

In the example of List 1, when the parameter texts of ***Console.WriteLine*** in the first two case blocks, which are "The first case." and "The second case." are selected successively by using the mouse or keyboard, the corresponding nodes of the element are need to be found in the CST firstly. Then, the two subtrees will be compared to extract the common nodes. Figure 2 and 3 show a subtree of each element in the CST respectively.

Though some nodes are omitted, it is not difficult to understand the structures and the position of each element, which is very obvious that those nodes in black frame are the same as the other subtree. The different nodes are just presented in red frame, and they have the same node type which is ***argument_list***.

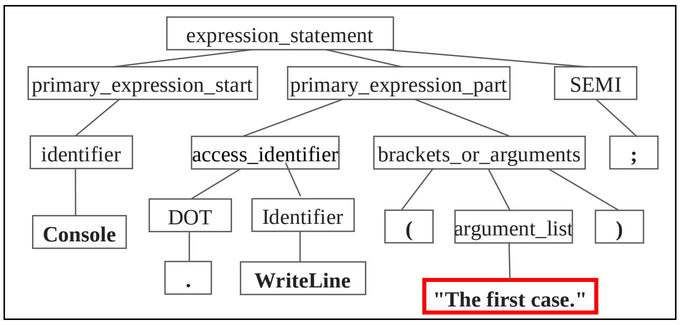


Fig. 2. A subtree of an example:
Console.WriteLine("The first case.");

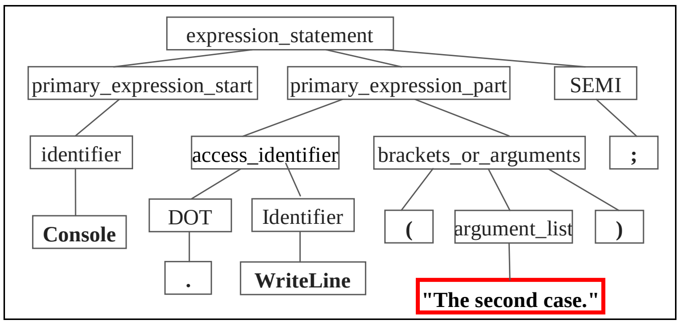


Fig. 3. A subtree of an example:
Console.WriteLine("The second case.");

3.2 Extracting the similarity pattern

As mentioned above, SimilarHighlight will compare the subtrees of two elements. It is very important to get valid subtree of each node corresponding to element, consisting of surrounding nodes: ancestor nodes, sibling nodes and descendant nodes. A data set of surrounding nodes of each node from parsed CST data will be extracted. The omitted data sets are presented like List 4 and 5. The numbers in the left are the index of data's in the data sets. As the index number is not consecutive, too many data are omitted to understand the relationship between the List 3 and List 4. Each data of the data set in List 4 is composed of the nodes type and nodes id and token text. Firstly, a traversal from the outmost node to token is presented as the data from the index 0 to 22.

Next, two methods will be used to find other surrounding nodes including: finding child nodes of all new nodes which are considered in other methods and finding sibling nodes of the direct parent node. And the methods will be used several times to collect more data to find similar elements more accurately. The program element texts in the expression can be found in bold. The common data of two data sets is the data set except the data of the index 24 and others omitted, and the number of common data is 52 in practice. It is obvious that the subtrees of two elements have the common nodes and the common data set instead of the common nodes are defined as the similarity pattern.

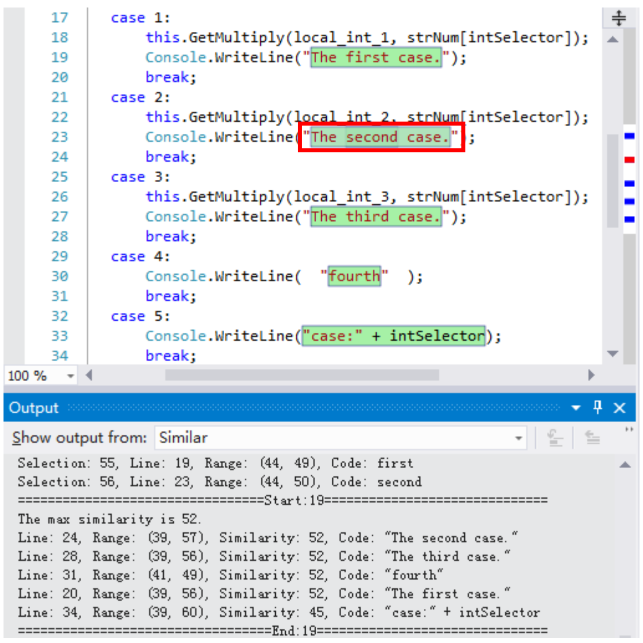
3.3 Extracting all possible matched elements

To ensure a high running performance, we can't traverse each program element in the file to judge whether it is a similar element. Lucky, the candidates can be extracted by using the effective node types of CST as mentioned above (3.1). In the example, they are both ***argument_list***. All elements that the effective node type is ***argument_list*** will be extracted as candidate elements and each of them will be compared with the similarity pattern. Then if they have the common data and the number of the common data is bigger than a threshold set before, the element will be seen as a valid match, in other words it is a similar element.

4. A visual studio extension

This approach is implemented in a visual studio extension called SimilarHighlight to evaluate the approach and implement it to help programmers increase their programming productivity. The main functions of the SimilarHighlight are as follows:

1) Highlight all the similar elements of last selected elements.
2) The previous or next similar element can be found by shortcut keys immediately, and the whole text of it will be selected to modify easily.
3) A margin will be added on the right side of the visual studio editor to offer relative position marks about similar elements.
4) A pane named "Similar" will be added into the output window, to offer more information about similar elements.
5) Some settings are provided to customize the tool, including enable or disable the functions, and similarity level which can change the threshold to increase or reduce the scope of similar elements.



```
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;
```

```
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. 4. The running result of SimilarHighlight

To present the functions of the tool, the running result of a more complicated example than the motivating example like Figure 4 is shown. In the parameter texts of ***Console.WriteLine*** in the first two case blocks: ***first*** and ***second*** the two words are selected successively by using the mouse or keyboard. It is very obvious that the similar elements look like being highlighted. Though the whole text in the token has the double quotations like ***"The first case."*** not ***The first case***, to modify quickly in the next operation, the double quotations will be ignored. The current cursor is located in the second case block whose background color is deeper than others. It can be found out the next similar element by pressing ***Ctrl + Alt + Right Arrow***, and the cursor will be changed to the next element, then the text of it can be modified immediately. As the results, many select and move operations will become unnecessary, and the minimal keystrokes are reduced.

Furthermore, another technique is worth mentioning that is to click a mark by left mouse button in the right margin, then the corresponding element of that mark will be selected, which is helpful to find an element jump some elements quickly. An additional function is that the *"**Similar**"* output window is used to offer selected element information and similar element information involves text and similarity in order. The similarity is the count of common data with similarity pattern. In this example, the max similarity is 52, while the similarity of the element in fifth case block is just 45, which is higher than the threshold is set before. In addition, though the part text of the element is selected by the mouse or keyboard, the element can be found exactly, if the source code in the file hasn't a serious format error.

5. Evaluation

To assess the effectiveness of this approach and SimilarHighlight, we conducted a set of experiments and compared the results against conventional methods. Specifically, it is necessary to investigate the following research questions:

RQ1: How can this tool to increase the programming productivity?

RQ2: Whether the tool is running smoothly?

5.1 Experiment 1

To investigate RQ1, the tool is used to accomplish some programming task in practice. An extreme example which consists of ten case blocks in a switch block like Figure 4 showed will be tested. (https://github.com/youfbi008/SimilarHiglight/blob/master/SimilarHighlight.Tests/SimilarityTest1.cs) As mentioned above, the third step of the Copy-Paste method that modifies the elements as expected to accomplish the task will be considered in this experiment.

	non-tool-using			tool-using		
	Time-cost	Key-strokes	Mouse clicks	Time-cost	Key-strokes	Mouse clicks
keyboard person	3min13s	540	4	2min5s	330	4
mouse person	2min26s	186	87	2min	260	16

Table 3. The running results in the extreme example

Table 3 show the time-cost, the keystrokes and the mouse clicks to present the programming productivity in non-tool-using and tool-using. It is obvious that using the tool can reduce 1/3 time-cost and keystrokes for a keyboard person in this example.

Therefore, a higher programming productivity can be obtained by using the tool especially for a keyboard person.

5.2 Experiment 2

The running performance about the tool will be considered because the line number of source code file the parsed xml text become too long with the line number of source code file, and the candidates are too many in some times.

To investigate RQ2, The average running time is measured in three selection patterns as follows:

Pattern 1: parameter texts of two methods are selected.

Pattern 2: names of two local variables are selected.

Pattern 3: names of two methods are selected.

		Pattern 1		Pattern 2		Pattern 3	
File Name	**SLOC**	**Count**	**ART**	**Count**	**ART**	**Count**	**ART**
CstInferrer	358	35	103ms	27	145ms	5	127ms
Line500	500	60	80ms	80	129ms	60	160ms
Line1000	1000	121	133ms	159	200ms	121	303ms
HLTextTagger	1053	85	150ms	94	163ms	17	268ms
Line5000	5000	605	351ms	795	845ms	605	1003ms

Table 4. The running results as the three patterns.

Table 4 shows the running results of selecting elements as the three patterns by using the tool. (https://github.com/youfbi008/SimilarHighlight) In this table, the names of test files and the source lines of code (SLOC) of the file, and the number of similar elements (Count), and average running time (ART) are showed to present the running performance of the tool.

As the table shows, these source files which SLOC is less than 5000 can be ran in 1 second. And the elements are highlighted earlier than Reference Highlighting of visual studio in practice [14], so programmers almost don't need wait the element be highlighted and continue the next operations. Furthermore, it is running by background thread.

Therefore, SimilarHighlight can run smoothly not to interrupt programmers continue the operations.

6. Related works

Nguyen presents an AST-based incremental approach that computes characteristic vectors for all subtrees of the AST for a file [15].

Murakami is conducting challenging research on automated code change and propose a technique to predict what kinds of program elements are deleted and added in the next change on Java methods [16].

Bruch propose intelligent code completion systems that learn from existing code repositories by searching for code snippets [17], and provide confidence values of the recommendations to default code completion widget.

7. Summary and future works

In this paper, we elucidated the problems in existing testing methods through motivating examples. We proposed an approach and developed a tool called SimilarHighlight to improve the problems. SimilarHighlight that suggests which program elements are similar to the last selected elements and might be modified at the next modifications. The elements will be highlighted and the text to be changed of the next element can be selected immediately by shortcut keys. Moreover, we evaluated the effectiveness of SimilarHighlight in empirical experiments.

As we presented, the tool can be used in programming task and modification task to increase the programming productivity. Furthermore, source code review is peer review of source code of computer programs. It is intended to find and fix defects overlooked in early development phases, improving overall code quality [18]. In our tool, the highlighting about similar elements can help reviewers find them easily, especially about the consistency checking.

In the future, we will improve our approach as follows.

1) Improve the running performance when the source lines of code are more than 5000.
2) Improve the precision to match the similar elements more effectively.
3) Support more programming languages.
4) Extract more patterns from programming habits

Reference

1) Huizhong Duan and Bo-June (Paul) Hsu. Online spelling correction for query completion. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 117-126, New York, NY, USA, 2011, ACM.
2) S. Han, D. R. Wallace, and R. C. Miller, “Code completion from abbreviated input,” in Proceedings, ASE. IEEE Computer Society, 2009, pp. 332–343.
3) M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In Proc. of ISESE 2004, pages 83–92, 2004.
4) I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In Proc. of ICSM ’98, pages 368–377, 1998.
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., 28(7):654–670, 2002.
6) B. Lagu ̈e, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In Proc. of ICSM ’97, pages 314–321, 1997.
7) Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In ICSE '08, pages 321{330, New York, NY, USA, 2008. ACM.
8) Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In SCAM, Montreal, October 2002. IEEE-CS.
9) Stefan Bellon , Rainer Koschke , Giulio Antoniol , Jens Krinke , Ettore Merlo, Comparison and Evaluation of Clone Detection Tools, IEEE Transactions on Software Engineering, v.33 n.9, p.577-591, September 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) T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen, "Scalable and incremental clone detection for evolving software," ICSM'09, 2009.
16) H. Murakami, K. Hotta, Y. Higo and S. Kusumoto. Towards Automated Code Evolution. IEICE technical report, Vol. 113 No. 422. pp. 107-112, Jan 2014.
17) M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In Proceegings of FSE, pages 213–222, Amsterdam, The Netherlands, 2009. ACM.
18) Uwano, H., Nakamura, M., Monden, A., and Matsumoto, K., "Analyzing individual performance of source code review using reviewers' eye movement", in Proceedings of 2006 symposium on Eye tracking research & applications (ETRA), San Diego, California, 2006, pp. 133-140.