A Study on
A Tool to Suggest Similar Program Element Modifications

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering

by

Yujian Yang
Student ID: 5112B079-0

Supervisor: Yoshiaki FUKAZAWA

July 26, 2014

Department of Computer Science and Engineering
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, I extracted all possible matching elements via similarity patterns from recently modified elements using a sub syntax tree comparison and then created a tool, SimilarHighlight. My 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, the tool supports C, C#, JAVA, and JavaScript, but in the future I will expand it to other languages.
## Contents

Abstract ......................................................................................................................... i

Contents ...................................................................................................................... ii

List of Lists ................................................................................................................... iii

List of Figures ............................................................................................................... iv

Introduction ................................................................................................................. 1

1.1 Keystroke Reduction ............................................................................................... 1

1.2 Selecting and Moving ............................................................................................. 2

1.3 Minimal Keystrokes ............................................................................................... 2

1.4 Continuous Similar Operations ............................................................................ 3

1.5 Similar Code and Code Clone Detection ............................................................... 4

1.6 Approach and Contributions .................................................................................. 5

Motivating Example .................................................................................................... 7

2.1 Copy-Paste Method ............................................................................................... 7

2.2 Representative Patterns ....................................................................................... 8

2.3 Conventional Methods .......................................................................................... 9

2.4 Problem Statement ............................................................................................... 11

SimilarHighlight ........................................................................................................ 13
3.1 Overview ............................................................................................................. 13
3.2 Parsing Source File ............................................................................................ 14
3.3 Determining Corresponding Subtree ................................................................. 17
3.4 Extracting Similarity Pattern .............................................................................. 20
3.5 Extracting Matching Elements ........................................................................... 24

Visual Studio Extension ............................................................................................ 26

4.1 Basic Functions .................................................................................................... 26
4.2 Optional Functions ............................................................................................... 27
4.3 Practical Example .................................................................................................. 28

Evaluation .................................................................................................................. 31

5.1 Experiment 1 ........................................................................................................ 31
5.2 Experiment 2 ........................................................................................................ 32
5.3 Experiment 3 ........................................................................................................ 34

Related Works ............................................................................................................ 37

6.1 About Keystroke Reduction .................................................................................. 37
6.2 About Code Highlighting ..................................................................................... 38
6.3 About Similar Code Detection .............................................................................. 38

Conclusions and Future Works .................................................................................. 41

Bibliography ............................................................................................................... 43
List of Lists

List 1: Three programming tasks in C# ................................................................. 3
List 2: First example of representative patterns in C# ......................................... 8
List 3: Second example of representative patterns in C# .................................... 9
List 4: Omitted xml text of the expression: Console.WriteLine("The first case.") ................................................................. 16
List 5: The pseudo code for determining outermost ancestor ............................. 18
List 6: Omitted node set of the element: "The first case." .................................... 21
List 7: The pseudo code for collecting surrounding nodes ................................. 22
List of Figures

Figure 1: Minimal keystrokes comparison by conventional methods .......... 12
Figure 2: Overview of SimilarHighlight ........................................ 14
Figure 3: Corresponding subtree of the element: "The first case." ............ 19
Figure 4: Omitted subtree of the expression: Console.WriteLine("The first case."); ................................................................. 20
Figure 5: Comparison of node sets to extract similarity pattern ................ 23
Figure 6: Extracting all possible matching elements .............................. 24
Figure 7: Running result of SimilarHighlight .................................... 29
Figure 8: Minimal keystrokes comparison ........................................... 32
Figure 9: Running results in the first task ........................................... 33
Figure 10: Running results in the second task ....................................... 34
Figure 11: Running results of the five patterns .................................... 36
Chapter 1

Introduction

Programming is a challenging job that often requires typing long codes via a keyboard. As a programmer counters his keystrokes in one year, that is 2.6 Million keystrokes. Hence, the reduction of necessary keystrokes in programming is very important and useful for programmers.

1.1 Keystroke Reduction

Many source code editors and tools such as Visual Studio and Eclipse are developed to help programmers to improve programming productivity. They are great because they have many great features about programming, debugging, testing and publishing to make the software development easy to accomplish. In the great features, code completion is one widely used productivity feature. Code completion 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 helps programmers decrease the number of keystrokes needed to save time spent typing
Moreover, the candidate suggestions can help programmers save time and reduce the errors because often the programmer 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].

1.2 Selecting and Moving

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. In this study, I focus on the operations about selecting and moving in code writing. To reduce the programming efforts, the necessary keystrokes need to be reduced.

1.3 Minimal Keystrokes

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]. Likewise, minimal keystrokes can be used to measure the programming effort when a programmer has a clear goal. Hence, programming effort should decrease when the number of minimal keystrokes is reduced, and programming productivity should increase as programming effort is decreased.
1.4 Continuous Similar Operations

Programmers are often faced with programming tasks of many continuous similar operations. List 1 shows three examples, 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. For these specific tasks, some

```csharp
    int intOne = 1;
    int intTwo = 2;
    int intThree = 3;
    int intFour = 4;
    int intFive = 5;

    //...... Ten local variables be initialized.
    string[] strNum = new string[] {
        "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten"
    }; // An array be initialized by explicitly setting ten elements.

    switch (intSelector)
    {
        case 111:
            this.GetMultiply(local_int_1, strNum[intSelector]);
            Console.WriteLine("The first case.");
            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;
        //...... A switch block has ten case block.
    }
```

List 1: Three programming tasks in C#
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 many similar code fragments in the source code.

1.5 Similar Code and Code Clone Detection

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].

About code clone, code clone analysis and removal tools are used to improve software quality and productivity like clone detection tools [26]. However, in code writing a tool to minimize the number of keystrokes during modifications of similar code does not exist.
1.6 Approach and Contributions

To improve programming productivity, I propose an approach to extract the similarity pattern from recently modified elements and provide all possible matching elements as modification suggestions for programmers. Similar to 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.

Research Questions

Specifically this work aims to answer three research questions.

RQ1: Does SimilarHighlight reduce the minimal keystrokes?

RQ2: Can SimilarHighlight tool improve the programming productivity?

RQ3: Does SimilarHighlight run smoothly without inconvenience?

Contributions

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.
Chapter 1. Introduction

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. It can be downloaded and to be installed in Visual Studio 2012.

The remainder of this paper is organized as follows. Chapter 2 provides a motivating example. Chapter 3 describes my proposed approach and tool, SimilarHighlight. Chapter 4 and 5 provide details and evaluate its functions, respectively. Chapter 6 describes related works. Finally, Chapter 7 is the conclusion and future work.
Chapter 2

Motivating Example

This section provides an example to demonstrate my approach and tool. Consider the switch block in List 2, which includes at least three case blocks, where each case block has a method with two parameters and a system output method.

2.1 Copy-Paste 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 values, 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. Otherwise, it needs almost twice the time to accomplish this task in code writing by hand. Hence, the more these similar operations, the higher efficiency obtained by the Copy-Paste method. Copy-Paste method as a very common activity is widely used in programming. However, sometimes it is not as good as its convenience (as mentioned in 1.5).
This study focuses on the third step of the Copy-Paste method: Modify the elements as needed to accomplish the task, which is similar to modification tasks that often occur in software maintenance and reengineering. In addition, in this study, a program 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.

2.2 Representative Patterns

Representative patterns of similar program elements are as follows:

**Pattern 1:** Method parameters (List 2).

**Pattern 2:** Case values of a switch (List 2).

```csharp
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 2: First example of representative patterns in C#
Pattern 3: **Array elements** (List 3).

Pattern 4: **Local variable names or values** (List 3).

Pattern 5: **Method names** (List 3).

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

List 3: Second example of representative patterns in C#

As basic code elements and grammars of programming languages, these are very common in source code.

### 2.3 Conventional Methods

The conventional methods for changing a text are broadly separated into two categories as follows.

(1) As is well-known, Backspace key can moves the display cursor one position backwards, deletes the character at that position, and shifts back the text after that position by one position. Therefore, to change a text, some people usually delete the characters of the text, and type the new characters.
Chapter 2. Motivating Example

(2) The other method is to select the whole text and type the new characters. This method is often known as a mouse select operation, but some people usually use this method to change a text efficiently by using a keyboard.

This study focuses on the second method which is more convenient and usually used by many programmers. When modifying 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].

**Mouse selection**

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. However, because there are more keyboard operations in code writing, the frequent switch between a mouse and a keyboard frequently is too much hassle for experienced programmers.

**Keyboard selection**

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 of the former method must be pressed fifteen times, and the latter one must be pressed four times. Then the number of minimal keystrokes for selecting this text is four. The latter method is significantly
better than the former one, but to select the period there must be to change the shortcut keys from [Ctrl]+[Shift]+[Right arrow] to [Shift]+[Right arrow].

2.4 Problem Statement

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, I 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 (Figure 1). Additionally, when elements are further separated in the code, more keystrokes are used to move between elements and find next target element, 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.
Figure 1: Minimal keystrokes comparison by conventional methods
Chapter 3

SimilarHighlight

A Tool to Improve Programming Productivity

3.1 Overview

I propose a tool (SimilarHighlight) to help programmers improve their productivity. The 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 summarizesthe main steps of SimilarHighlight.

Firstly, 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 of this CST. Two different elements of the last selected elements are picked to as the comparison objects. They are compared to extract the common surrounding node set as a similarity pattern. In addition, candidate node type is also extracted to determine the candidate nodes.

Secondly, each of the candidate nodes is compared to the similarity pattern to
Finally, SimilarHighlight highlights all the corresponding elements of the matching nodes and presents them to the programmers.

### 3.2 Parsing Source File

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**

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. It is an open source project which I am contributing to, and it will be improved for my tool.

**Node and Subtree**

A program element is usually represented as a single node in the generalized syntax tree. However, a program element in this CST is represented as a subtree, which consists of multiple nodes, including a token node. Each node has some properties such as a node type and a node id.

Node type is the type of a node; it is used to describe the feature of the node, it may be seen as the type of a subtree whose top node is this node to describe the feature of this subtree. Node type may be token, identifier, argument_list, statement, etc.

Node id is automatically generated by Code2Xml, to be used to differentiate from the nodes which have the same node type but have different structure. In addition, if the node is a token node, it also has positional data to describe the position of the element in the source code.

**Parsed XML**
Chapter 3.  SimilarHighlight

As an easy-to-understand example, List 4 shows a parsed xml where the xml 

```
<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_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" endline="86" endpos="37">
                  WriteLine
                </TOKEN>
              </IDENTIFIER>
            </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">
                    "The first case."
                  </STRINGLITERAL>
                </argument_list>
              </arguments>
            </brackets_or_arguments>
          </access_identifier>
        </primary_expression_part>
      </primary_expression>
    </expression_statement>
  </statement>
```

List 4: Omitted xml text of the expression: `Console.WriteLine("The first case.")`
texts for `Console.WriteLine("The first case.")` is omitted. Although the complete xml text is triple times longer, the main elements such as `Console`, `WriteLine`, "The first case.", and some symbols such as (, ), and . in this expression are presented in red. The parsed XML text contains the complete information of the source code, and in other words, the all elements can be found in this CST. Worth mentioning is that "The first case." including double quotation marks is a single program element as a parameter.

In my 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` which are presented in bold. If the selected text is an expression such as `Console.WriteLine("The first case.")` or a phrase of a method such as `Console.WriteLine`, the corresponding node can also be determined by positional data. It is obvious that the corresponding node is not a token; it will be an ancestor node of a subtree which contains the complete information of the selected text. According to List 4, the corresponding node of `Console.WriteLine` may be `primary_expression`, while the corresponding node of `Console.WriteLine("The first case.")` may be `expression_statement`.

### 3.3 Determining Corresponding Subtree

To represent the corresponding subtree of the current element, outermost ancestor also must be determined. Though the corresponding node like token mentioned above contains some information of an element, but it is not complete in this CST.
Outermost Ancestor

The outermost ancestor is outermost one in the ancestors which is ancestor of a node and has no other immediate child nodes. In addition, in practice a token node sometimes has some sibling nodes to recode the blank spaces in this CST. Therefore when the corresponding node is a token node, the parent node is still considered.

List 5 shows a pseudo code to present my approach in determining outermost ancestor. Then the corresponding subtree can be represented by the nodes from the outermost ancestor to the current node. Figure 3 shows the omitted corresponding subtree for "The first case." and the node types used to describe the nodes. The complete subtree contains ten nodes in practice.

<table>
<thead>
<tr>
<th>SET corresponding node of an element to Innermost Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE parent of Innermost Node exist AND</td>
</tr>
<tr>
<td>the count of child nodes of the parent of Innermost Node is 1 OR</td>
</tr>
<tr>
<td>Innermost Node is a token node</td>
</tr>
<tr>
<td>DO</td>
</tr>
<tr>
<td>SET the parent of Innermost Node to Innermost Node</td>
</tr>
<tr>
<td>ENDWHILE</td>
</tr>
</tbody>
</table>

SET Innermost Node to Outermost Ancestor Node

List 5: The pseudo code for determining outermost ancestor
Element Type

As the node type is seen as the type of a subtree, and a subtree represents an element in my CST, hence the outermost ancestor type is seen as the type of an element. The element type is used to extract the candidates. The outermost node type of "The first case." is argument_list, hence the type of "The first case." is argument_list.

Because an expression can be seen as an element in this approach, Figure 4 shows the corresponding subtree of the expression: Console.WriteLine("The first case."); as an element. Although some nodes are omitted, the structure and position can be understood, especially about "The first case." element.

According to the subtree of Console.WriteLine("The first case.");, the next step considers the subtree to determine the surrounding nodes of "The first case." element.
3.4 Extracting Similarity Pattern

In the example of List 2, 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 to extract similarity pattern.

**Surrounding Nodes**

The surrounding nodes generally consist of ancestor nodes, sibling nodes, and descendant nodes. Because it is important to effectively collect this information, my approach extracts the surrounding nodes from CST. This information is then used to construct a node set.
Node Set

List 6 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 4 and 6. In practice, the node type, node id, and token text shown in List 4 are used to construct the data of the node set in List 6. The main elements of the expression such as Console, WriteLine, and "The first case." marked in red can be found in this node set, hence the main elements are seen as the surrounding nodes as would be expected.

```
[0]:<argument_list>
[1]:<argument_list><argument>
[2]:<argument_list><argument><argument_value><expression>
......
[22]:<argument_list><argument><argument_value><expression>......<primary_expression_start><literal><STRING_LITERAL>
[23]:<argument_list>
[24]:argument_list>"""The first case.""
[25]:argument_list-'
[26]:argument_list-'
......
[28]:argument_list-TOKENS<literal>
[29]:argument_list-RPAREN<literal>
......
[40]:argument_list<arguments><brackets_or_arguments><access_identifier>
[41]:argument_list<arguments><brackets_or_arguments><access_identifier>"WriteLine"
[42]:argument_list<arguments><brackets_or_arguments><primary_expression_part><primary_or_array_creation_expression>
......
[56]:argument_list<arguments><brackets_or_arguments><primary_expression_part><primary_or_array_creation_expression>
```

List 6: Omitted node set of the element: "The first case."
Algorithm

List 7 shows a pseudo code to present my 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.

| 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 7: The pseudo code for collecting surrounding nodes
Chapter 3. SimilarHighlight

Similarity Pattern

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

Node set of "The first case."

[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)
......

Node set of "The second case."

[23]:argument_list280
[24]:argument_list>"The second case."
[25]:argument_list-'('
[26]:argument_list-')'
[28]:argument_list-TOKENSchar_literal279(
  [29]:argument_list-RPARENchar_literal281)
......

Figure 5: Comparison of node sets to extract similarity pattern
3.5 Extracting 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 element type (as mentioned in 3.1). Figure 6 shows the process to determine similar elements.

![Diagram showing the process of extracting all possible matching elements](image)

Figure 6: Extracting all possible matching elements

**Candidate Node Type**

Because the types of the two elements are both `argument_list`, hence `argument_list` is seen as a candidate node type. Then all nodes where the
Chapter 3. SimilarHighlight

outermost node type is *argument_list* are extracted as candidate nodes, then 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). The matched element is similar element.

Finally, similar elements are highlighted based on the positional data of the corresponding token nodes.
Chapter 4

Visual Studio Extension

SimilarHighlight is implemented in a visual studio extension to evaluate my approach and to help programmers improve their programming productivity. The functions of the SimilarHighlight consist of basic functions and optional functions.

4.1 Basic Functions

Element Highlighting

Elements similar to the last selected element are highlighted. This function is similar to Syntax highlighting and Reference Highlighting [14] can help programmers find something expected.

Element Selecting

The previous or next similar element can be found immediately via shortcut keys, and the whole text is selected for easy modification. More specifically, when Ctrl + Alt + Right Arrow keys are pressed, the cursor will be move to the next highlighted element, and the whole text of that element is selected. Then the
programmer types a new character to make it change. Likewise, Ctrl + Alt + left Arrow keys can make the previous highlighted element to be selected. In addition, Escape key can remove the current highlighting.

### 4.2 Optional Functions

**Maker bar**

A marker bar is added on the right side in the visual studio editor to offer relative position markers of similar elements. This function is similar to the marker bar in Eclipse IDE. If the document is long, a marker tells the programmer where he needs to scroll in order to see a particular object. Therefore, this function is important to remind programmers some elements need to be modified, especially they are far from current cursor.

**Output Pane**

A pane named “Similar” is added into the output window to provide more information about similar elements. Some helpful information such as the positional data and the texts of similar elements are presented and sometimes to help programmers know at a glance.

Others information such as the number of max similarity is a count of common data in the similarity pattern (as mentioned in 3.3). And the similarity number of each similar element is used to present their similarity between this element and similarity pattern. The similarity number will be modified to present a percentage of the max similarity in soon.
Setting Page

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. Such as the color settings about highlighted elements and makers in the marker bar can be changed. The similarity level has three levels which are high, middle and low.

4.3 Practical Example

To present the functions of the tool, Figure 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. As the results, although each of the selected texts is only part of parameter text, SimilarHighlight can locate them and find out the corresponding tokens in CST accurately. In addition, 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 + Right Arrow. Then the text of the new element is selected, and 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 marker bar to select the corresponding element of that mark. This allows a 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. In this example, the maximum similarity is 52. The similarity of the element in fifth case block is 45, which exceeds the predetermined threshold. These functions mentioned above can work if the source code in the file does not have a serious format error.

Figure 7: Running result of SimilarHighlight
Similar to a clone detection tool, SimilarHighlight also find similar code fragment not just a single token. However, the main function of this tool is highlighting similar elements while programmers are still writing codes. Finding the similar fragment will takes more time, hence it is more appropriate as a static detection function. Moreover, similar code fragment often occurs in the different source files not in the same file, but the highlighting function is only valid in the current window to offer suggestions to programmers.
Chapter 5

Evaluation

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

5.1 Experiment 1

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

Figure 8 compares the minimal keystrokes for the five similar element patterns and the reduction rates. My 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.
5.2 Experiment 2

To investigate RQ2 (Can SimilarHighlight tool improve the programming productivity?), I 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 List 2, which includes pattern 1 and pattern 2 (https://github.com/youfbio08/SimilarHighlight/blob/master/SimilarHighlight.Tests/SimilarityTest1.cs) This experiment tests the third step of the Copy-Paste method (element modification). I measured the time and the keystrokes necessary to accomplish each task with and without SimilarHighlight. The results were compared to calculate the reduction rates. It will not have a beneficial effect if I use my 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 tasks, respectively. The averages of time usage and the number of keystrokes were calculated to determine the reduction rate in using SimilarHighlight. SimilarHighlight reduces the coding time by approximately 23% and the keystrokes by 44% in the first task (Figure 9). Because the reduction in keystrokes is nearly twice the reduction in time, as programmers become more familiar with this tool, the time reduction should become larger.

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

Figure 9: Running results in the first task
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 my method and conventional methods was statistically significant based on *wilcoxon signed-rank tests* (p-value < 0.05) [25]. Therefore, my method is significantly better than conventional methods.

### 5.3 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 SimilarHighlight is considered. It is possible that this tool does not run smoothly or is inconvenient to programmers. To investigate RQ3 (Does
SimilarHighlight run smoothly without inconvenience?), I conducted an experiment in which SimilarHighlight 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 SimilarHighlight 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. Moreover, I found some reasons which cause the running performance lowly. The most reason is the algorithm for determining the corresponding token of current selected element in CST. For example, two similar program elements are located at the top and bottom of the source code. Then select them respectively and record time for determining the corresponding token in background thread. I found that the time is significantly difference, and the bottom one even needs double of the top one. To solve this problem, I will improve my tool in soon.
Figure 11: Running results of the five patterns
Chapter 6

Related Works

There are different types of related works about my tool from different angles. I briefly summarize them in the following categories: keystroke reduction, code highlighting, similar code detection. I will discuss the existing methods and the difference between my tool and existing methods respectively.

6.1 About Keystroke Reduction

As mentioned in 1.1, code completion is a great technique, which reduce a great deal of keystrokes on code writing to help programmers improve programming productivity. Code completion is helping programmers on program element writing. Besides this, code refactoring help programmers to decrease programming effort such as the keystrokes in some programming and modification tasks, although it is generally known as the practice of applying behavior-preserving changes to existing code, can enhance the quality of software systems. The source code is changed automatically by code refactoring tools.

In contrast, my tool reduces the keystrokes for selecting text and moving cursor in a continuous manual modification to help programmers improve programming
productivity.

6.2 About Code Highlighting

As the default functionality of mainstream source code editors, **Syntax highlighting** is one strategy to improve the readability and context of the text. The reader can easily ignore large sections of comments or code, depending on what they are looking for. It also helps programmers find errors in their program. Besides this, **Reference Highlighting** help programmers find all instances of the current symbol in the document.

Similar to the two highlighting functions, my tool help programmers find the similar program elements which might be modified in next modifications. It is important for preventing missing some necessary modifications, because the missed modifications sometimes cost a lot to find out.

6.3 About Similar Code Detection

The main aim of my approach is to detect similar code, which is also the main aim of code clone detection. There is a huge body of existing works related to code clone detection techniques and tools. Four main approaches, namely string-based, token-based, tree-based and PDG-based, are used to analyze source code to detect similar code.

**String-based**

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. A code fragment matches another if both fragments are contiguous sequences of source lines with some consistent identifier mapping scheme. This approach may be applied to various languages and the semantics of the underlying programming language is completely ignored.

**Token-based**

Kamiya et al. [5] provide a token-based code clone detection tool named CCFinder, which focuses on analyzing large-scale systems with a limited amount of language dependence. It transforms tokens of a program according to a language-specific rule and performs a token-by-token comparison. This approach also provides a visualization tool that allows visual recognition of matches within large amounts of code.

**Tree-based**

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]. To avoid the complexity of full subtree comparison, Koschke et al. [27] use alternative tree representations. In that approach, AST subtrees are serialized as AST node sequences to construct a suffix tree. This method allows find syntactic clones at the speed of token-based techniques.

My approach is also parse-tree-based. However, because of the different aim, my approach uses the subtree comparison to extract a similarity pattern, then to find out the similar program elements. To improve the running performance, I will
refer to the approach about suffix tree. To make my tool to be used widely, I will improve it on detecting similar code fragments in future.

**PDG-based**

PDG is program dependency graph which is a representation of a program that represents only the control and data dependency among statements [17]. This representation abstracts from the lexical order in expressions and statements, which are semantically independent. The search for similar codes is then turned into the problem of finding isomorphic subgraphs. Krinke et al. [24] uses the PDG-based method to detect maximal similar subgraphs.

Due to the different aims, my approach is to find similar code in one source file, not in the entire project, and to finds mainly the similar program elements not the code fragments. In addition, my tool, SimilarHighlight is a productivity tool. It suggests the programmer to modify them at the next modifications and reduce the keystrokes to improve the programming productivity.
Chapter 7

Conclusions and Future Works

I 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, I evaluated the effectiveness of SimilarHighlight in empirical experiments.

SimilarHighlight 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.

My aim is to make SimilarHighlight the default functionality of the source code editor. In the future, I will improve my approach and 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 SimilarHighlight. Generally modification history of similar elements will be considered to infer the next element to be modified. In addition, my tool is mainly used in detect similar program elements now. I will improve it on detecting the code fragments like code clone detection tools.

• Support more programming languages. Currently SimilarHighlight can be used in C, C#, JAVA, JavaScript, and PHP files. I am 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, I intend to extract potential modification patterns. Additionally, instead of highlighting all of the text of an element, only the part to be modified will be highlighted.

• 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.
Bibliography


