Comment-based Keyword Programming

Yusuke Sakamoto, Haruhiko Sato, Satoshi Oyama, Masahito Kurihara

***Abstract*—Keyword programming is a technique to generate code fragments automatically from the several keywords provided by the user and the context of the code. It reduces a burden of remembering the details of a particular language or API. In this paper, we extend the technique for using comments, instead of keywords, in the search of expressions. The use of comments is expected not only to motivate us to write comments frequently, but also to give us more semantic information which is useful for improving quality of generated codes.**

***Index Terms*—Autocomplete, Code assistants, Code completion**

I. Introduction

With the advancement of software development technology, a lot of programming languages such as Java, C#, VB, PHP, Ruby, etc. exist today. However, modern programmers must learn and remember the details of these many programming languages and APIs. It is a heavy burden for programmers to learn, remember, and use these language syntaxes correctly. Keyword programming [1] is a technique to reduce the burden of remembering the details of a particular language or API. It uses a few keywords provided by the user to search for expressions that are possible in the context of the code.

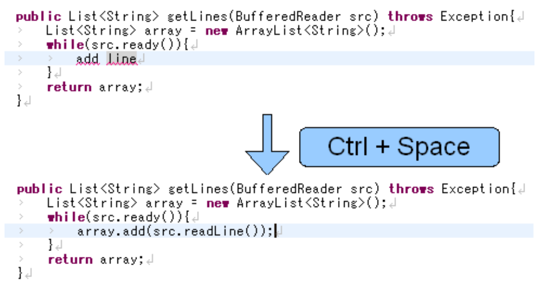
In this paper, we extend the keyword programming and present the idea of comment-based keyword programming, which provides a tool to create executable code fragment from a single line comment input by the user programmer. Thus it can further reduce the burden of programmers. Programmers generally have to write comments explaining what an executable code means for making it easier to understand. Comment-based keyword programming utilizes their works of writing comments for creating candidates of an executable code fragment.

The paper is organized as follows. In Section II, we briefly review the general idea of the keyword programming. In Section III, we present the main idea of the comment-based keyword programming. In Section IV, we conclude our work and discuss some future work.

II. Keyword Programming

In keyword programming [1], the user provides a few keywords for the input query to search for a desired code fragment. The user interface takes the form of a code completion interface in an IDE. For instance, Fig.1 shows a user entering **add line** in a Java file, which the system translates in-place to **array.add(src.readLine())**. The generated expression contains the user's keywords **add** and **line**, but also fills in many details, including the receiver objects array and src, the full method name readLine, and the formal Java syntax for method invocation. Keyword programming is related to work on searching for examples in a large corpus of existing code. It can be distinguished by the kind of query provided by the user. In Prospector [4] and XSnippet [3], these systems require types for the input. However, in keyword programming, the input is not restricted to a type.

Manuscript received January 12, 2011; revised February 7, 2011. The authors are with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo 060-0814, Japan. E-mail: sayuu@complex.eng.hokudai.ac.jp.



```
public List<String> getLines(BufferedReader src) throws Exception{
    List<String> array = new ArrayList<String>();
    while(src.ready()){
        add line
    }
    return array;
}
```

```
public List<String> getLines(BufferedReader src) throws Exception{
    List<String> array = new ArrayList<String>();
    while(src.ready()){
        array.add(src.readLine());
    }
    return array;
}
```

Fig. 1. In keyword programming, the user types some keywords, presses a completion command (such as Ctrl-Space in Eclipse), and the keywords are translated into a valid expression

A. Model

Keyword programming models the available methods, fields and local variables in the case of Java. Although it could be applied to many languages, this paper focuses on Java. Keyword programming defines the model M as the triple (T, L, F), where T is a set of types, L is a set of labels used for matching the keywords, and F is a set of functions.

1) Type set: T: Each type is represented by a unique name. For Java, keyword programming gets this from the fully qualified name for the type. Examples include int and java.lang.Object.

2) Label set: L: Keyword programming uses labels to represent method names, so that keyword programming can match them against the keywords in a query. To get the keywords from an identifier, keyword programming breaks up the identifier at punctuation and capitalization boundaries. For instance, the method name currentTimeMillis is represented with the label (current, time, millis).

3) Function set: F: Functions are used to model each component in an expression that keyword programming wants to match against the users keyword query. In Java, these include methods, fields, and local variables. Keyword programming models expressions generated by a keyword query as a function tree. Each node in the tree is associated with a function from F, and obeys constraints of the types of return value and parameters.

B. Evaluation of a Function Tree

Given a function tree where T_{funcs} is the list of functions in the tree, keyword programming calculates the score as follows:

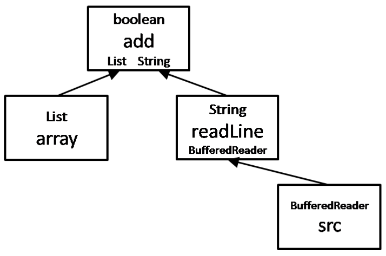


Fig. 2. This tree is a function tree representing the Java expression `array.add(src.readLine())` from Fig.1. Each node is associated with a function. The words at the top of each node represent the return type, and the words at the bottom represent the parameter types.

- $+1.0$ for each keyword k in the query where there exists an $f \in T_{funcs}$ such that $k \in label(f)$. This gives 1 point for each keyword that the tree explains.
- -0.05 for each $f \in T_{funcs}$. This favors function trees with fewer functions.
- For each $f \in T_{funcs}$, consider each $w \in label(f)$, and subtract 0.01 if w is not in the query. This favors less verbose function trees.
- $+0.001$ for each $f \in T_{funcs}$ where f is a local variable, or f is a member variable or member method of the enclosing class. This favors functions and variables which are close to the user's context.

Since it takes a lot of time to calculate all the combinations of the functions, keyword programming uses a dynamic programming (Fig.3) to reduce the amount of calculation.

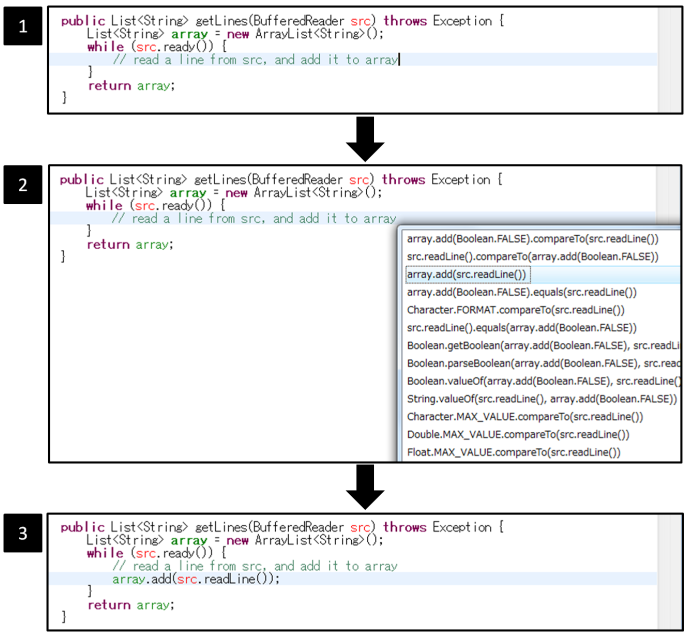
Algorithm 1 Pseudocode to fill out the dynamic programming table in keyword programming

```
procedure DynamicProgram()
for each $1 \leq i \leq h$ do
  for each $t \in T$ do
    $bestRoots(t, i) \leftarrow \emptyset$
    for each $f \in F$ where $ret(f) \in sub(t)$ do
      $e \leftarrow GetBestExplForFunc(f, i-1)$
      if $e < -\infty$ then
        $bestRoots(t, i) \leftarrow bestRoots(t, i) \cup (e, f)$
      end if
      $bestRoots(t, i) \leftarrow GetBestN(bestRoots(t, i), r)$
    end for
  end for
end for

procedure $GetBestExplForFunc(f, h_{max})$
$e_{cumulative} \leftarrow expl(f)$
for each $p \in params(f)$ do
  $e_{best} \leftarrow (-\infty, 0, 0, 0, \ldots)$
  for each $1 \leq i \leq h_{max}$ do
    for each $(e', f') \in bestRoots(p, i)$ do
      if $e_{cumulative} + e' > e_{best}$ then
        $e_{best} \leftarrow e_{cumulative} + e'$
      end if
    end for
  end for
  $e_{cumulative} \leftarrow e_{best}$
end for
return $(e_{cumulative})$
```

III. Comment-based Keyword Programming

For utilizing the comments in generating code fragments in keyword programming, we propose comment-based keyword programming. The actual flow of comment-based keyword programming is as follows: first, the user input a short, one line comment and invokes the tool by command of Ctrl-Space. Second, the tool shows multiple candidates of executable code fragment on a pop-up window and the user select a suitable one. Finally, the tool outputs the code and the input comment remains on the source code (Fig.4).



```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        // read a line from src, and add it to array
    }
    return array;
}
```

```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        // read a line from src, and add it to array
    }
    return array;
}
```

```
array.add(Boolean.FALSE).compareTo(src.readLine())
src.readLine().compareTo(array.add(Boolean.FALSE))
array.add(src.readLine())
array.add(Boolean.FALSE).equals(src.readLine())
Character.FORMAT.compareTo(src.readLine())
src.readLine().equals(array.add(Boolean.FALSE))
Boolean.getBoolean(array.add(Boolean.FALSE), src.readLi
Boolean.parseBoolean(array.add(Boolean.FALSE), src.read
Boolean.valueOf(array.add(Boolean.FALSE), src.readLine()
String.valueOf(src.readLine(), array.add(Boolean.FALSE))
Character.MAX_VALUE.compareTo(src.readLine())
Double.MAX_VALUE.compareTo(src.readLine())
Float.MAX_VALUE.compareTo(src.readLine())
```

```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        // read a line from src, and add it to array
        array.add(src.readLine());
    }
    return array;
}
```

Fig. 4. This picture shows a flow of the actual works of our tool. It is a plug-in of the Eclipse IDE.

A. Advantage

If you use our tool at programming, there are two advantages. Since the input is a comment, you unify the two works, writing codes and writing comments. Since the input is a comment, you tend to write more comments naturally than usual. Our tool reduces the works of programming and enhances the readability.

B. Algorithm for multiple outputs

We have improved the algorithm of previous study presented in the section of algorithm. The algorithm of previous study outputs only one candidate. We have improved it in order to output multiple candidates. The improved points are as follows:

- Each cell in the dynamic programming table keeps a few hundred of candidates. In previous work, each cell contains at most 3 candidates.
- Each cell keeps a function tree, instead of the root function of the tree.
- Outputs are a few hundred of function trees whose height is at most 3 (same as previous study) and having desired return type.

The reason that the number of candidates is limited to a few hundred of pairs is the limit of memory size (4GB in

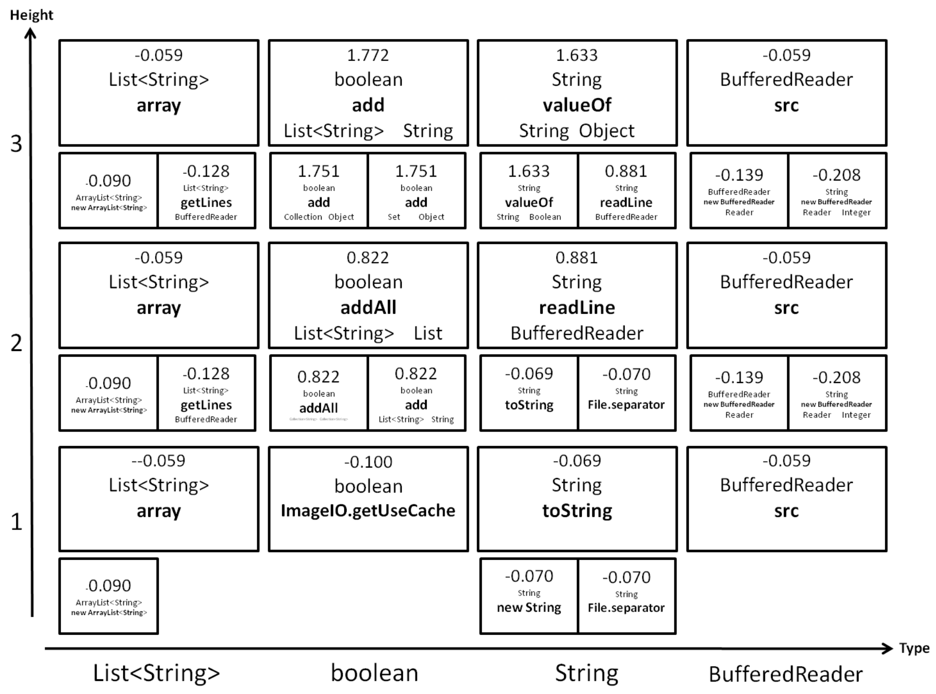

Height
3
-0.059 List<String> array
1.772 boolean add List<String> String
1.633 String valueOf String Object
-0.059 BufferedReader src
0.090 ArrayList<String> new ArrayList<String>
-0.128 List<String> getLines BufferedReader
1.751 boolean add Collection Object
1.751 boolean add Set Object
1.633 String valueOf String Boolean
0.881 String readLine BufferedReader
-0.139 BufferedReader new BufferedReader Reader
-0.208 String new BufferedReader Reader Integer
2
-0.059 List<String> array
0.822 boolean addAll List<String> List
0.881 String readLine BufferedReader
-0.059 BufferedReader src
0.090 ArrayList<String> new ArrayList<String>
-0.128 List<String> getLines BufferedReader
0.822 boolean addAll
0.822 boolean add List<String> String
-0.069 String toString
-0.070 String File.separator
-0.139 BufferedReader new BufferedReader Reader
-0.208 String new BufferedReader Reader Integer
1
--0.059 List<String> array
-0.100 boolean ImageIO.getUseCache
-0.069 String toString
-0.059 BufferedReader src
0.090 ArrayList<String> new ArrayList<String>
-0.070 String new String
-0.070 String File.separator
Type
List<String>
boolean
String
BufferedReader

Fig. 3. This is the dynamic programming table for a run of the algorithm on the input add line in the context of Fig. 1. Each row represents a height, and each column represents a return type. Each cell contains up to three boxes. The box with the highest score in each cell is enlarged. Each box represents a function with the name in the center, the return type above the name, and the parameter types listed below the name. The return type of each function must be a subtype of the column type that the box is in. The number above the return type is the score of the function.

our environment). Although this algorithm clearly consumes larger amount of memory than previous algorithm, our tool responds within a second even if the size of function set F is a few thousand.

Algorithm 2 Our DynamicProgram procedure

```
procedure DynamicProgram()
for each $1 \leq i \leq h$ do
    for each $t \in T$ do
        $bestRoots(t, i) \leftarrow \emptyset$
        for each $f \in F$ where $ret(f) \in sub(t)$ do
            $tree \leftarrow GetBestExplForFunc(f, i-1)$
            $e \leftarrow tree.getRoot().getE()$
            if $e < -\infty$ then
                $bestRoots(t, i) \leftarrow bestRoots(t, i) \cup tree$
            end if
            $bestRoots(t, i) \leftarrow GetBestN(bestRoots(t, i), r)$
        end for
    end for
end for
```

In addition to output multiple candidates, we have tried to consider the use of frequency of the selected candidate. Because of higher probability of occurrence, a lot of candidates is useful. However, a lot of candidates is also a burden for users when selecting what they need. We think that this is a similar problem of web search engines. Like web search engines, the list of candidates presented to users should be customized for each user using the tool.

We define a sorting rule for user customized list. We calculate the score of each candidate of function tree by the sum of (1) the number of times of the use of a function tree and (2) the score for a function tree calculated the rules

Algorithm 3 Our GetBestExplForFunc procedure

```
procedure $GetBestExplForFunc(f, h_{max})$
$e_{cumulative} \leftarrow expl(f)$
$tree_{best} \leftarrow CreateFunctionTree(f, e_{cumulative})$
for each $p \in params(f)$ do
    $e_{best} \leftarrow (-\infty, 0, 0, 0, \ldots)$
    $tree_{param} \leftarrow null$
    for each $1 \leq i \leq h_{max}$ do
        for each $tree \in bestRoots(p, i)$ do
            if $e_{cumulative} + e' > e_{best}$ then
                $e_{best} \leftarrow e_{cumulative} + e'$
                $tree_{param} \leftarrow tree$
            end if
        end for
    end for
    $e_{cumulative} \leftarrow e_{best}$
    $tree_{best} \leftarrow AddChild(tree_{best}, tree_{param})$
end for
return $(e_{cumulative})$
```

explained in the section of II-B. The candidate list is sorted in descending order of the score.

C. Implementation

We have implemented our idea as an Eclipse plug-in. Our tool works following steps:

1) A programmer inputs keywords on the editor, and invokes our tool.
2) Our algorithm of keyword programming generates an output of a few hundred of candidates taking source code context into account.

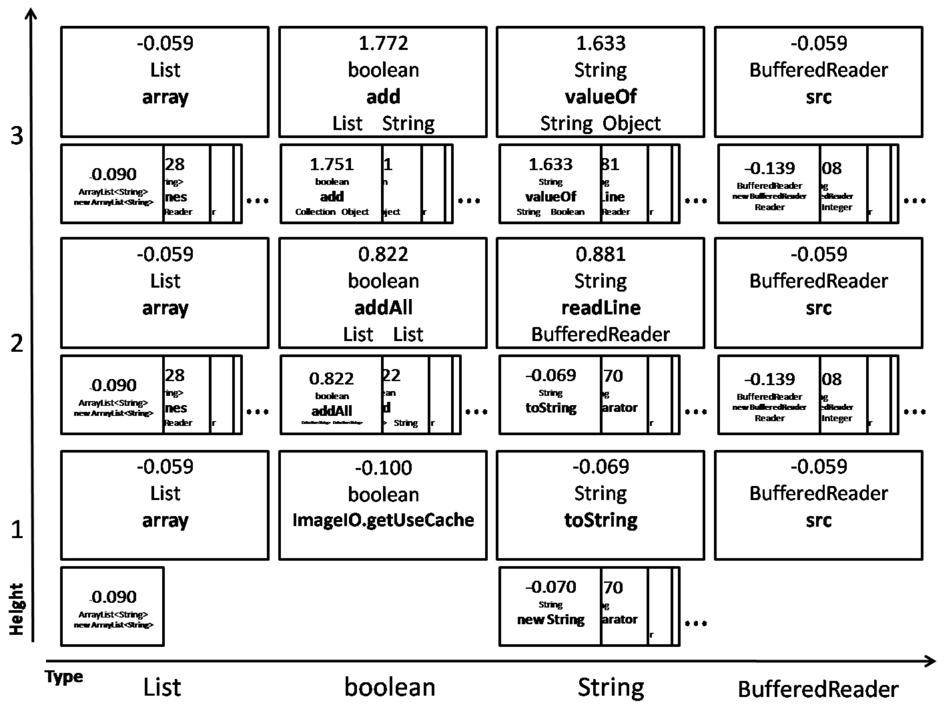

Height
3
2
1
Type
List
boolean
String
BufferedReader
-0.059 List array
1.772 boolean add List String
1.633 String valueOf String Object
-0.059 BufferedReader src
0.822 boolean addAll List List
0.881 String readLine BufferedReader
-0.100 boolean ImageIO.getUseCache
-0.069 String toString
0.090 ArrayList<String> new ArrayList<String>
-0.070 String new String

Fig. 5. This figure is an image of the improved dynamic programming table in order to output multiple candidates. Each cell of the table is a function tree. Each cell contains a few hundred of function trees.

3) Our tool sorts list of output candidates with their use of frequency.
4) The programmer selects candidate and the use of selected candidate is written in the database.

Fig.6 shows the usefulness of considering the use of frequency. These two pictures are the snapshots after invoking our Eclipse tool on the same context. The left snapshot is the first time of the invocation. The right is the third time. In this context, the candidate that user needs are colored in the list. The candidate exists higher in the list of the right snapshot. The introduction of the use of frequency makes commonly-used candidates easy to find.

IV. Conclusion and Future Work

One line short comment is often used line by line especially on a commercial source code that requires high readability. Comment-based keyword programming is very useful for such situations. Our future work is applying a very easy natural language processing for processing input small comment (Fig.7). We think that applying it will improve the precision of the tool. We will analyze parts of speech of the input using part-of-speech tagger and apply the naming conventions of programming. For example, the input: "**add line**" is a Verb-Object pattern. The result is likely to be "**add(line)**" or "**line.add(X)**" or "**addLine()**". The input: "**robot moves to (x,y)**" is a Subject-Verb pattern. The result is likely to be "**robot.move(x,y)**" or "**move(robot,x,y)**". The object and the subject are likely to be a child node of the verb in a function tree. We will add a new scoring rule to existing four rules on the basis of this way of thinking. The other future works are as follows: We will extract important words for programming by analyzing open source project forums with TF/IDF to take priorities of words into account. Since keyword programming cannot handle synonyms like "**add**" and "**append**", we will allow our tool to handle synonyms by making a thesaurus specialized for software developments.

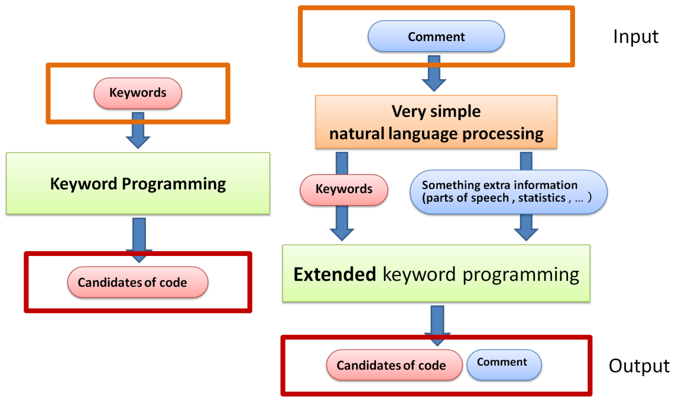
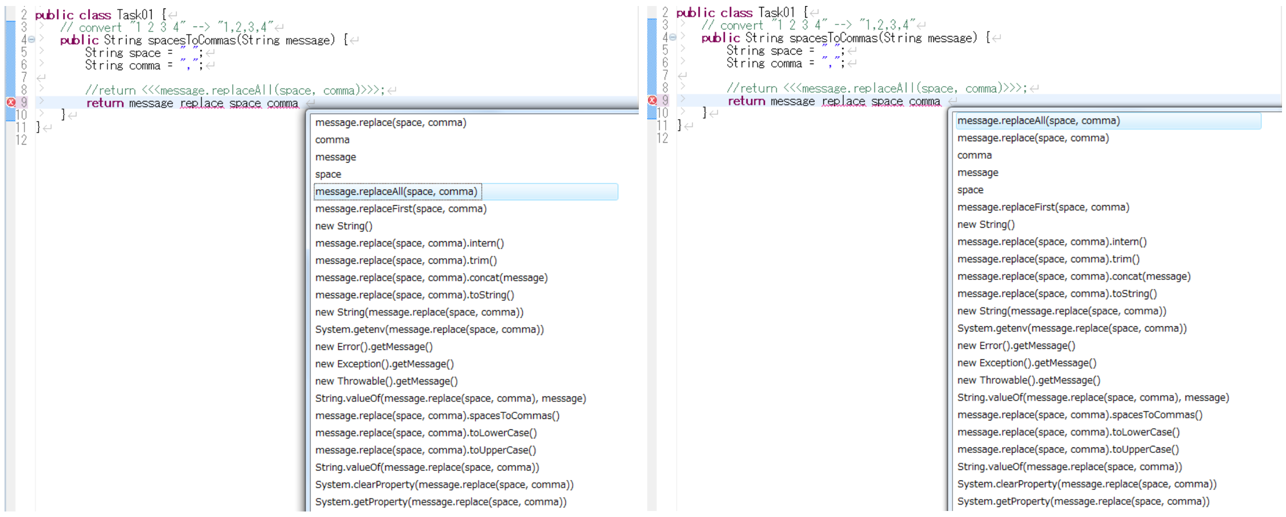

Input
Comment
Keywords
Very simple natural language processing
Keyword Programming
Keywords
Something extra information (parts of speech , statistics , ...)
Candidates of code
Extended keyword programming
Candidates of code
Comment
Output

Fig. 7. These are the overviews of the tool. The left is keyword programming and the right is our study.

References

[1] G. Little, R. C. Miller. Keyword programming in Java. *Automated Software Engineering* , 16(1), pp. 37-71, 2009.
[2] Steven P. Reiss. Semantics-based code search *Proceedings of the 31st International Conference on Software Engineering* , pp. 243-253, 2009.
[3] N. Sahavechaphan, K. Claypool. XSnippet: Mining For Sample Code. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2006)* , pp. 413-430.
[4] D. Mandelin, L. Xu, R. Bodik, D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* , pp. 48-61.
[5] Rosco Hill, Joe Rideout. Automatic Method Completion. *Proceedings of the 19th IEEE international conference on Automated software engineering* , pp. 228-235, 2004.
[6] Brad A. Myers, Andrew J. Ko, Sun Young Park, Jeffrey Stylos, Thomas D. LaToza, Jack Beaton. More natural end-user software engineering. *Proceedings of the 4th international workshop on End-user software engineering* , pp. 30-34, 2008.



```
public class Task01 {
    // convert "1 2 3 4" --> "1,2,3,4"
    public String spacesToCommas(String message) {
        String space = " ";
        String comma = ",";

        //return <<<message.replaceAll(space, comma)>>>;
        return message replace space comma
    }
}
```

```
message.replace(space, comma)
comma
message
space
message.replaceAll(space, comma)
message.replaceFirst(space, comma)
new String()
message.replace(space, comma).intern()
message.replace(space, comma).trim()
message.replace(space, comma).concat(message)
message.replace(space, comma).toString()
new String(message.replace(space, comma))
System.getenv(message.replace(space, comma))
new Error().getMessage()
new Exception().getMessage()
new Throwable().getMessage()
String.valueOf(message.replace(space, comma), message)
message.replace(space, comma).spacesToCommas()
message.replace(space, comma).toLowerCase()
message.replace(space, comma).toUpperCase()
String.valueOf(message.replace(space, comma))
System.clearProperty(message.replace(space, comma))
System.getProperty(message.replace(space, comma))
```

```
public class Task01 {
    // convert "1 2 3 4" --> "1,2,3,4"
    public String spacesToCommas(String message) {
        String space = " ";
        String comma = ",";

        //return <<<message.replaceAll(space, comma)>>>;
        return message replace space comma
    }
}
```

```
message.replaceAll(space, comma)
message.replace(space, comma)
comma
message
space
message.replaceFirst(space, comma)
new String()
message.replace(space, comma).intern()
message.replace(space, comma).trim()
message.replace(space, comma).concat(message)
message.replace(space, comma).toString()
new String(message.replace(space, comma))
System.getenv(message.replace(space, comma))
new Error().getMessage()
new Exception().getMessage()
new Throwable().getMessage()
String.valueOf(message.replace(space, comma), message)
message.replace(space, comma).spacesToCommas()
message.replace(space, comma).toLowerCase()
message.replace(space, comma).toUpperCase()
String.valueOf(message.replace(space, comma))
System.clearProperty(message.replace(space, comma))
System.getProperty(message.replace(space, comma))
```

Fig. 6. Snapshots of the simulation using our Eclipse plug-in. Although these two pictures are on the same context, the item that user needs (colored items in the list) exists in different order. The left picture is the first time of the command invocation. The right is the third time. By the introduction of the use of frequency, commonly-used items are way up on the list.

[7] Brad A. Myers, John F. Pane, Andy Ko. Natural programming languages and environments. *Communications of the ACM* , 47(9), pp. 47-52, 2004.

[8] Miller, George A. "WordNet - About Us." WordNet. Princeton University, http://wordnet.princeton.edu. 2009.