Improving code completion based on repetitive code completion operations

Takayuki Omori, Hiroaki Kuwabara, Katsuhisa Maruyama

Although code completion is inevitable for effective software development, existing code completion tools can be improved. A previous study noted that developers sometimes perform ineffective repetitions of the same code completion operations. Hence, this paper introduces a statement, “A more recently inserted code completion candidate should be given a higher rank among previously inserted items in the candidate list.” To confirm this statement, this paper examines the following three points. First, an experiment using operation histories is presented to reconfirm that developers more frequently repeat recent code completion operations. Second, a tool called RCC Candidate Sorter is presented. It alters the sorting algorithm of the default Eclipse code completion tool to prioritize candidates inserted by recent code completion. Finally, an experiment is conducted to evaluate the performance of the tool. The experimental result shows that it outperforms an existing method.

1 Introduction

Code completion is inevitable for effective software development. Every integrated development environment (IDE) provides a code completion tool as far as we know. If a developer types a part of an identifier and launches the code completion tool, the tool typically generates a list of possible candidate identifiers beginning with the typed characters. Then, the developer can easily input the whole identifier by selecting the appropriate one from the candidate list. Thus, a code completion tool helps reduce key strokes and improves productivity during source code editing. Eclipse, a de-facto IDE for Java, provides a set of functions called content assist, which includes the code completion function. Murphy et al. reported actual Eclipse usage by developers [14]. They asserted that all experimental target developers used content assist, which is the fifth most commonly used function.

There is no doubt that state-of-the-art code completion tools greatly contribute to real software development. However, existing code completion mechanisms can be improved. Our research focuses on repetitions of code completion. Repetitive text editing is a traditional problem. Several methods have been proposed to reduce the burden of repetitive text editing [3] [9] [10]. However, repetitive code completion is not well investigated.

In our previous work [18], we examined how code completion operations (cc-operations) are performed using developers’ operations recorded in the Eclipse Java editor. We focused on repetitions of cc-operations and performed a preliminary experiment, which demonstrated that recently inserted identifiers by cc-operations are repeated more frequently than other identifiers. Here, the word “repeat” means that a cc-operation inserts the same string as a previous one. In the paper, a repetitive cc-operation (RCC) is defined as a cc-operation whose inserted string is the same as the preceding or following cc-operation within the target operation history. Because repeating the same editing operation is tedious and ineffective, RCCs should be streamlined to reduce the burden on developers. Moreover, RCCs have high potential to be exactly predicted as code completion candidates since they
have far less variations than other candidates.

In this work, we introduce a statement, “A more recently inserted code completion candidate should be given a higher rank among previously inserted items in the candidate list.” If this statement is true, a code completion tool that sorts based on when each item was inserted can improve code completion. The main contribution of this paper is confirming the statement. To confirm it, this paper describes the following three points:

1. An experiment reconfirms that cc-operations tend to be repeated with shorter time intervals. To extend the experimental generality and reliability, the experiment employs a larger dataset than our previous work [18].

2. A tool called RCC Candidate Sorter (RCC-CS) is implemented. This tool sorts code completion candidates provided by the default Eclipse code completion tool (ECC) based on the insertion time of each candidate in the preceding cc-operation.

3. An experiment that evaluates the performance of RCC-CS by comparing the existing method [19]. It indicates that RCC-CS can be applied to actual software development.

In this paper, a code completion strategy based on our statement is called a repetitive strategy, in contrast to a traditional strategy like ECC. RCC-CS adopts a repetitive strategy. We do not claim that the traditional strategy should be simply replaced by a repetitive strategy, but rather that factors to improve the traditional strategy and build better code completion tools should be explored.

The rest of the paper is organized as follows. Section 2 describes about ECC and a motivating example. Section 3 presents the experimental settings, including our dataset of operation histories, the terminology in the paper, and how to extract the cc-operations from operation histories. Section 4 describes an experiment to show that RCCs occur more frequently with shorter time intervals. Section 5 presents RCC-CS and its ranking performance based on our dataset. Additionally, RCC-CS is compared with an existing tool to demonstrate the applicability of a code completion tool based on a repetitive strategy. Section 6 provides several discussions, including threats to validity, while Section 7 explains related works. Finally, Section 8 concludes the paper with a short summary and future work.

2 Code completion overview

In this section, we explain code completion in Eclipse†1, since it is a commonly used IDE in Java programming and our tool is based on it. The explanation is based on the default settings†2 of Eclipse 4.3.1.

2.1 Code completion in Eclipse

Without code completion, identifiers in source code must be entered character by character. This method is prone to typos and is burdensome on developers. To avoid such inefficiencies, Eclipse provides a code completion tool (ECC). While a developer is editing source code in the Eclipse Java editor, ECC can be explicitly invoked by hitting Ctrl+Space keys. If the developer manually types a part of an identifier as a prefix and launches code completion, possible candidate identifiers starting with the prefix characters are shown as a list. Then the developer can input the whole identifier by selecting the appropriate one from the candidate list. Code completion can also be automatically invoked, such as by inserting a member access operator. If no prefix is typed, ECC shows a list including all possible candidate identifiers.

Because the candidate list contains many items, an effective ranking algorithm to sort the candidates is very important. The ECC ranking algorithm considers the kinds (e.g., class name, method name) and the types (e.g., int, double, String) of candidates and the context where it is invoked. We explain the details with an example of a code completion candidate list in Figure 1. The developer explicitly invokes code completion after typing "int a = " . In this case, any candidate whose type is int is prioritized based on type estimation in the current code. Moreover, within a group of identifiers matching the type, local variables, fields, and methods are placed in this order. If candidates

†1 Additional explanation about Eclipse code completion function is available at http://help.eclipse.org/kepler/topic/org.eclipse.jdt.doc.user/tips/jdt.tips.html.

have the same type and kind, they are sorted in alphabetical order (like localA, localB, and localC). Any candidate with a different type is subsequently listed. In addition, ECC can insert code templates. For example, when the developer enters "new" and invokes code completion, the prefix is replaced with a statement that creates a new object. Candidates derived from code templates follow single-identifier candidates.

2.2 Motivating example

Unfortunately, ECC does not learn from developers' operation histories. Thus, it presents the same candidate list regardless the recent developers' use of code completion. This can be sometimes frustrating for developers. Figure 2 shows an example of repetitive code completion†3. In the figure, NormalOperation class contains many public members: getDeletedText(), getDeveloper(), getFile(), getInsertedText(), toString(), toString(int), etc. Two NormalOperation instances (no1 and no2) are declared as arguments of isSame() method. In this example, a developer is writing an if-statement in the method. After he/she entered "no1.getInsertedText().compareTo()" (using code completion for getInsertedText()), he/she typed "no2.". Then, ECC automatically generates the list shown in Figure 2. In this situation, the developer intends to input getInsertedText() to compare two inserted strings. However, ECC ranks getDeletedText() at the top due to its ranking algorithm denoted in Section 2.1. The candidate selection can be shifted by hitting the up and down arrow keys. In this case, the developer must hit the down arrow key three times to select the fourth item and then press the enter key or take his/her hand off the keyboard to double-click the fourth candidate. If the correct item is placed at the top, it can be inserted by pressing the enter key once.

Often a developer repeats the same code completion operation within a short time interval. However, the traditional tool does not leverage such repetitions.

3 Experimental settings

This section presents the details of experimental settings. Since we use our experimental dataset in both reconfirmation of the RCC observation (Section 4) and the evaluation for our proposed tool (Section 5), we explain the dataset in this section. Section 3.1 provides a general description of our dataset. We define several kinds of operation sets and metric values in Section 3.2. Section 3.3 presents how code completion data are obtained from our dataset.

3.1 Dataset

We collected operation histories during actual software development using OperationRecorder [17]†4. The seven target projects are shown in Table †4 The current version of OperationRecorder records operations into XML files. It can record more
Table 1 Target projects

<table>
<thead>
<tr>
<th>Developer</th>
<th>NOP</th>
<th>NCOP</th>
<th>NRCC</th>
<th>NREP</th>
<th>Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Author 1</td>
<td>60,508</td>
<td>1,586</td>
<td>1,013</td>
<td>1,948</td>
<td>230</td>
</tr>
<tr>
<td>B Author 2</td>
<td>23,757</td>
<td>368</td>
<td>174</td>
<td>180</td>
<td>197</td>
</tr>
<tr>
<td>C Author 2</td>
<td>34,339</td>
<td>429</td>
<td>197</td>
<td>168</td>
<td>225</td>
</tr>
<tr>
<td>D Author 2</td>
<td>6,488</td>
<td>116</td>
<td>57</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>E Author 2</td>
<td>15,271</td>
<td>246</td>
<td>76</td>
<td>62</td>
<td>38</td>
</tr>
<tr>
<td>F Student 1</td>
<td>4,463</td>
<td>168</td>
<td>103</td>
<td>180</td>
<td>21</td>
</tr>
<tr>
<td>G Student 2</td>
<td>9,606</td>
<td>220</td>
<td>118</td>
<td>144</td>
<td>102</td>
</tr>
<tr>
<td>Total</td>
<td>154,432</td>
<td>3,133</td>
<td>1,738</td>
<td>2,746</td>
<td>—</td>
</tr>
</tbody>
</table>

1. Each project was engaged by one of four developers; two are authors of this paper and the other two are master’s course students majoring in computer science. We used a partial operation history with regard to projects B, C, and G since they started before the recording tool was completed. The metric values in Table 1 are explained in Section 3.2. The new dataset has over 2.8 times\(^5\) the number of operations (NOP) of the former data in [18]. In addition, 55.5\(^6\) of cc-operations extracted from the new dataset are involved in cc-operation repetitions. Given this ratio, RCCs should not be ignored when considering how to improve the code completion strategy.

Although the operation history contains several kinds of operations, the three described here pertain to this paper.

- **Normal operation**
  This kind of operation indicates a normal insertion, deletion, or modification of text. It is derived from a developer’s manual code edit. It contains inserted and/or deleted strings and the offset value and the file name where the operation was performed. It corresponds to successive keystrokes, not a single keystroke. For example, if a developer types “a”, “b”, and “c”, and hits the backspace key, two normal operations are published; the first one contains “abc” as its inserted string and the second one contains “c” as its deleted string.

- **Compound operation**
  This kind of operation indicates an operation that is automatically performed by content assist or other functions (e.g., automated refactorings). It contains a label indicating the kind of the operation. It also contains normal operations as its children, which present actual change of source code by the automatic edit. Code completion is performed by replacing a prefix string with a whole identifier. Thus, when such a prefix exists, the child of a compound operation corresponding to code completion indicates deleting the prefix and inserting the identifier.

- **Menu operation**
  This kind of operation indicates an invocation of a menu function. A command invocation via its shortcut key also causes this kind of operation. It has a label indicating the command ID of the menu item.

Each operation contains information about the time when the operation was performed.

### 3.2 Terminology

In this section, we define several terms and metric values used in this work. First, we provide a formal definition of an RCC.

Here we introduce the notation for the target operation history. Let \(o_i\) be an arbitrary operation within the operation history. The operation history \(H\) is defined as

\[
H = \{o_i | 1 \leq i \leq m\}
\]

where \(m\) is the number of operations within \(H\). It can be also denoted as \(|H| (= m)\).

As shown in Section 3.3, cc-operation groups are
extracted from the target operation history. Each cc-operation group includes a single compound operation and other kinds of operations.

Let $C$ be a set of compound operations within cc-operation groups, which is denoted as

$$ C = \{ c_i \in g_i \mid 1 \leq i \leq n \} $$

where $c_i$ is a compound operation included in an extracted cc-operation group $g_i$. The number of cc-operation groups included in $H$ is $n$.

Here, we consider a pair of compound operations. The pair of $c_i$ and $c_j$ is represented as follows.

$$ p_{(i,j)} = (c_i, c_j) $$

$P$, which is the set of pairs of compound operations within $C$, is represented as

$$ P = \{ p_{(i,j)} \mid c_i.time < c_j.time, c_i \in C, c_j \in C \} $$

where $c_i.time$ indicates the time when the compound operation $c_i$ was performed.

Within a pair, the former operation is called the precedent cc-operation, and the latter is called the succedent cc-operation; the precedent cc-operation always precedes the succedent one in time order. The time interval of pair $p_{(i,j)}$ is defined as

$$ p_{(i,j)}.intv = c_j.time - c_i.time $$

where $c_i$ is the precedent cc-operation of pair $p$, and $c_j$ is the succedent one.

Two sets ($RccPairs$ and $NRccPairs$) are represented as follows. Here, $c_k.ins$ indicates a string inserted by the compound operation $c_k$. Texts are compared based on their content. The condition for $RccPairs$ is equal to having a return value of zero for `java.lang.String.compareTo()` when $c_i.ins$ and $c_j.ins$ are the parameters. $NRccPairs$ has the opposite condition.

$$ RccPairs = \{ p_{(i,j)} \in P \mid c_i.ins = c_j.ins \} $$

$$ NRccPairs = \{ p_{(i,j)} \in P \mid c_i.ins \neq c_j.ins \} $$

An RCC denotes a cc-operation belonging to a pair within $RccPairs$. Every RCC appears in one or more pairs within $RccPairs$ and every cc-operation belonging to a pair within $RccPairs$ is an RCC.

Next we introduce several metric values regarding the operation history and RCCs.

**NOP:** The number of operations within the target operation history, which is calculated as $|H|$.

**NCOP:** The number of extracted cc-operations within the target operation history, which is calculated as $|C|$.

**NRCC:** The number of RCCs. NRCC is normally less than double the value of $|RccPairs|$ since a single cc-operation may belong to multiple pairs within $RccPairs$.

**NREP:** The number of pairs of cc-operations with the same inserted string, which is calculated as $|RccPairs|$.

**NNREP:** The number of pairs for every cc-operation except for repeated ones, which is calculated as $|NRccPairs|$. $NREP + NNREP$ equals the number of pairs for every cc-operation.

Figure 3 shows an example calculation of the above values (except for NOP) using five cc-operations ($c_1, \ldots, c_5$) in an operation history. The character written in each circle corresponds to the inserted string of the operation. Here, the inserted strings of $c_1, c_3,$ and $c_5$ are the same. Hence, NCOP and NRCC for the operation history are 5 and 3, respectively.

**RccPairs** and **NRccPairs** for this example are represented as follows.

$$ RccPairs = \{ p_{(1,3)}, p_{(3,5)}, p_{(1,5)} \} $$

$$ NRccPairs = \{ p_{(1,2)}, p_{(1,4)}, p_{(2,3)}, p_{(2,4)}, p_{(2,5)}, p_{(3,4)}, p_{(4,5)} \} $$

Therefore, NREP and NNREP are calculated as 3 and 7, respectively.

### 3.3 Extracting code completion data

The dataset presented in Section 3.1 includes all operations recorded by OperationRecorder. Here we explain how information of cc-operations is extracted from the dataset.
3.3.1 Dividing the operation history into sessions

First, the whole operation history of a target project is divided into development sessions. Because we assumed that a developer’s intention for an operation does not carry over long breaks, herein a blank period of 30 minutes (or more) between operations is deemed as a session boundary.

3.3.2 Removing needless operations

As explained in Section 3.1, an operation history includes kinds of operations. First, all operations except for normal, compound, or menu operations are removed from the operation history. Moreover, any compound operation whose label does not match "Typing" is filtered out since all compound operations corresponding to code completion have this label. Similarly, any menu operation whose label does not match "org.eclipse.ui.edit.text.contentAssist.proposals" is also deleted.

3.3.3 Extracting cc-operation groups

A fragment of an operation history is denoted by a bracketed sequence of initial characters corresponding to the kinds of operations in this paper. For example, [NMC] means a sequence of a normal operation, a menu operation, and a compound operation in this order. In a cc-operation group, [N] corresponds to an insertion of the prefix string. [M] corresponds to an explicit invocation of code completion, while [C] corresponds to the candidate insertion.

Even after the filtering denoted in Section 3.3.2, compound operations that are not derived from cc-operations still remain. To remove such needless operations, operation fragments that match [MC], [NMC], and [MNC] are extracted as cc-operation groups because most cc-operations in our dataset conform to these patterns. Prior to determining whether to apply the patterns, we considered using a state transition model of ECC (denoted in Appendix A). However, the state transition model cannot perfectly determine the cc-operations from recorded operations. In general, the number of [N] and [M] associated with a single cc-operation cannot be determined, so the boundary of a cc-operation group is sometimes ambiguous. Moreover, the cancellation of code completion cannot be precisely detected since it does not appear in the operation history.

A compound operation corresponding to a cc-operation replaces the prefix (the part of the identifier that a developer typed) with the whole identifier. The replaced text is held as the deleted string of the compound operation. Therefore, when a cc-operation group is detected, whether the deleted string matches the tail of the inserted string of the previous normal operation is verified.

3.3.4 Normalizing the textual representation

Occasionally the formats of real inserted strings differ in compound operations, leading to a mismatch in the textual comparison between the same identifier insertions. Therefore, the following process is used to normalize the inserted strings:

- Removing ephemeral strings and jointing successive insertions
  Initially, two successive operations are combined if the latter one deletes the entire string inserted by the former one. If both the inserted and deleted strings of the resultant operation are empty, the operation is removed.
- Removing operator characters
  Operator characters, such as a dot, following identifiers are sometimes included in the inserted string. These are deleted.
- Removing method parameters
  ECC automatically inserts tentative method parameters when a developer selects a candidate method. Therefore, the inserted strings for overload methods differ. However, we assume that most developers are not concerned about the parameter type when selecting a method name from the candidate list. Hence, the parameters are removed, leaving only the parentheses.
- Removing type parameters
  ECC generates the proper type parameters when they can be predicted. They do not hinder a developer’s operations since all the developer has to do is to insert the identifier itself. Therefore, type parameters are unnecessary in our experiment and they are removed.
- Removing characters that inhibit a canonical comparison
  The extra white space characters and any characters not used in the identifiers or keywords are removed.
4 Analyzing the time interval of RCCs

This section presents an experiment to reconfirm the result in our previous work regarding the time intervals of RCCs.

In the preliminary experiment regarding RCCs [18], we found that the frequency of RCCs is higher for a shorter time interval between the two cc-operations. In this paper, we reconfirm the result with our new dataset to extend generality and reliability of the result.

We settled with time interval classes of 15 seconds in length. That is, the time intervals of cc-operation pairs belonging to the first time interval class are less than 15 seconds. Though we also examined class lengths of 30- and 60- seconds, overall tendency of time-series RCC distribution is almost the same regardless the class length. Consequently, hereafter the class length is 15 seconds.

We calculated NREP and NNREP for each time interval class. Figure 4 shows the results where the primary y-axis indicates the number of pairs on a log scale and the secondary y-axis indicates the rate \( \frac{\text{NREP}}{\text{NREP} + \text{NNREP}} \) for each class. The x-axis is the time interval class indices up to 30 minutes time interval (the 120th class) since classes with longer time intervals only include insubstantial data. Shorter time interval classes have more cc-operations and RCCs. Moreover, the rates also have the same tendency, indicating that the time-series distribution of RCCs is uneven; the rates are more frequent for shorter time interval classes.

We examined the significance of the difference of RCC frequency for each time interval class to estimate in which classes the frequency is significantly high. Here, we used Pearson’s chi-squared test since we assumed the index of a time interval class is a nominal scale. The null hypothesis is “NREP and NNREP are even regardless of time intervals”. If the null hypothesis is rejected in a class, the sample NREP in the class is expected to superior to the corresponding expected NREP.

Table 2 shows the result for the first 12 classes. In the Expected column, the proportion of NREP to NREP+NNREP for every class indicates the proportion of total NREP to total NREP+NNREP (3.32%). As the p-value shows, the null hypothesis was rejected with respect to all classes with a time interval of under 135 seconds with 99% significance level. However, the null hypothesis was not rejected for most of the other classes (rejected in 25 scattered classes, not rejected in 805 classes).

As the result of the above analysis, we conclude that the frequency of RCCs in shorter time interval classes is significantly higher. It should be noted that 135 seconds is not the definite threshold to determine the RCC frequency because the statistical analysis depends on the sample dataset, significance level, and the time interval class length. Currently, we have not attained a criterion for this threshold.

Given the result, we concluded that a recently inserted code completion candidate should be placed at a higher rank among previously inserted items in the code completion candidate list.

5 Improving code completion rankings based on RCCs

This section presents RCC-CS†7, which is an implementation of a repetitive strategy. It also presents a comparative experiment to evaluate the ranking method of RCC-CS.

5.1 RCC Candidate Sorter

To clarify how much a repetitive strategy can improve code completion, we built RCC-CS, which

\[ \text{Fig. 4 NREP and NNREP in time-interval classes} \]
sorts the code completion candidates that ECC provides based on the most recent insertion time. That is, a candidate which is inserted by the most recent cc-operation is placed at the top of the candidate list. The history of identifier insertions is discarded when a developer closes the IDE. Therefore, candidates that have never been inserted after the IDE launch are not prioritized.

Figure 5 shows a screenshot of code completion by RCC-CS for the same scenario shown in Figure 2. In this example, every method call for no1 and no2 was inserted by cc-operations. The last code completion for a NormalOperation instance inserted getInsertedText(). Therefore, getInsertedText() is placed at the top in the candidate list. In contrast, ECC shows them in alphabetical order as shown in Figure 2. Since the candidates are generated by ECC, the candidate list does not include identifiers out of the current scope.

Note that RCC-CS is not intended to exclude ECC. We suppose that the developer can switch ECC and a code completion tool with a repetitive strategy as needed. That is, the developer uses the latter only when performing an RCC. Otherwise, the default (ECC) is used. We think this selection can be performed appropriately in most cases since RCCs occur very often right after the precedent cc-operations as mentioned in Section 4.

### 5.2 Ranking comparison

Evaluating the ranking mechanism of a code completion tool is difficult. A user experiment is a considerable candidate for evaluation. Generally, existing tools and a novel tool are compared in such an experiment. Although there are many methods to improve existing code completion mechanisms (see Section 7), most of them are not provided as running tools on Eclipse. So, it is hard and resource-consuming to set up a user experiment. Moreover, we could not find any existing research focusing RCCs in the development environment which we focused. This means that we can obtain no data derived from existing tools used in performing RCCs. Therefore, we cannot claim efficiency of our method by simply comparing experimental results of previous papers and ours.

Still, we compare ranking results by Robbes’s method [19] (baseline) and our tool to indicate whether RCC-CS can be useful enough in actual software development. The reasons why we chose this baseline are as follows: (1) The baseline paper provides details of ranking performance and accu-

---

**Table 2 Comparison of NREP and NNREP**

<table>
<thead>
<tr>
<th>Intv. class</th>
<th>Sample</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NREP</td>
<td>NNREP</td>
</tr>
<tr>
<td></td>
<td>rate</td>
<td>NREP</td>
</tr>
<tr>
<td>[0, 15)</td>
<td>250</td>
<td>1784</td>
</tr>
<tr>
<td>[15, 30)</td>
<td>162</td>
<td>1303</td>
</tr>
<tr>
<td>[30, 45)</td>
<td>135</td>
<td>1113</td>
</tr>
<tr>
<td>[45, 60)</td>
<td>81</td>
<td>1073</td>
</tr>
<tr>
<td>[60, 75)</td>
<td>75</td>
<td>927</td>
</tr>
<tr>
<td>[75, 90)</td>
<td>65</td>
<td>926</td>
</tr>
<tr>
<td>[90, 105)</td>
<td>46</td>
<td>759</td>
</tr>
<tr>
<td>[105, 120)</td>
<td>39</td>
<td>738</td>
</tr>
<tr>
<td>[120, 135)</td>
<td>39</td>
<td>720</td>
</tr>
<tr>
<td>[135, 150)</td>
<td>29</td>
<td>742</td>
</tr>
<tr>
<td>[150, 165)</td>
<td>34</td>
<td>713</td>
</tr>
<tr>
<td>[165, 180)</td>
<td>39</td>
<td>735</td>
</tr>
</tbody>
</table>
racy rates by their tool. (2) The baseline tool exploits developers’ operations similar to our tool.

The baseline paper introduced quite high quality code completion mechanisms. So, we think that a tool excelling the baseline has sufficient applicability to today’s actual development.

Here, we have to explain about differences between the baseline experiment and ours. First, the baseline tool runs on Squeak IDE and the target language is Smalltalk. The different language conventions may affect the experimental results. Second, the prediction targets of the baseline tool are only method and class names, whereas our tool predicts any kinds of repetitive insertions by cc-operations. Finally, our dataset (described in Section 3) and the baseline dataset differ. We cannot use the baseline dataset due to its format and target language.

Since our dataset contains all editing operations, including cc-operations, we can virtually rebuild the code completion candidate list the tool generates, but the list is incomplete. Several recording problems prevent restaging of the actual code completion. For example, OperationRecorder does not record project properties, such as build paths. So, we cannot restore class names out of the developer’s project. Instead, we used the virtual candidate list that is derived from the list of cc-operations in our dataset. The virtual candidate list simply includes identifiers that were inserted by past cc-operations. We present an example of generating the virtual candidate list. Here, imagine a developer inserted the following identifiers by cc-operations in this order in the same development session.

(1) getDeveloper(), (2) getString(), (3) NormalDatum, (4) NormalDatum, (5) StringComparer, (6) getDeveloper()

Here, we focus on the situation where the developer inserts (6) (the second insertion of getDeveloper()). When the prefix length is 0, the virtual candidate list is as follows.

(1) StringComparer, (2) NormalDatum, (3) getstring(), (4) getDeveloper()

The order of identifiers is complied with reversed time order of past insertions. The two occurrences of NormalDatum are abbreviated to one occurrence. So, the correct candidate is ranked fourth.

If the developer types "g" as the prefix, the virtual list changes as follows.

(1) getstring(), (2) getDeveloper()
Table 3 Ranks in candidates

<table>
<thead>
<tr>
<th>Rank</th>
<th>Prefix length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>69.88</td>
</tr>
<tr>
<td>2</td>
<td>17.20</td>
</tr>
<tr>
<td>3</td>
<td>6.51</td>
</tr>
<tr>
<td>4</td>
<td>1.78</td>
</tr>
<tr>
<td>5</td>
<td>1.16</td>
</tr>
<tr>
<td>6</td>
<td>0.89</td>
</tr>
<tr>
<td>7</td>
<td>0.18</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.09</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
</tr>
<tr>
<td>11-</td>
<td>2.32</td>
</tr>
</tbody>
</table>

When the prefix is longer than the candidate, the candidate is placed at the first rank.

6 Discussion

6.1 Repetition factors

We discuss factors of RCCs based on our experiments. Though the factors described in this section are not exhaustive, they give a part of reasons why the repetitive strategy can improve code completion since RCCs derived from the factors mentioned here are performed in shorter time intervals in most cases.

6.1.1 Java language specifications and coding conventions

We observed that a part of Java language specifications causes an RCC. For example, Java generics tend to cause recurrent type names. The following is an example code fragment of generics. Java programs often include patterns similar to this.

```java
Collection<T> col = new Collection<T>();
T item = col.get(i);
```

In the code fragment, T indicates a type name. This example includes three type name appearances. ECC can correctly predict the second appearance on inserting the constructor invocation, so, in most cases, a developer writing this code would repeat cc-operations twice for the type name.

Hunt and Thomas classified repetitive representations caused by programming languages as imposed duplication [7]. We cannot avoid operations inserting them as well.

We claim that the burden of RCCs should be reduced. However, we do not claim that repetitions of the same code should be reduced. This is because a repetitive representation sometimes contributes to readability. For example, in the following code fragments, it is easier to know the type of a member of col from the second code than the first one, though it requires an extra RCC.

```java
col.get(i).getValue();
```

```java
T item = col.get(i);
item.getValue();
```

6.1.2 Cohesion of a developers’ task

A programming task has an associated vocabulary set [19]. Identifiers included in the vocabulary set are often used and tend to cause repetitions in the context.

While a developer is working on a specific function associated with a vocabulary set, edit operations on multiple references to the same identifier related to the function tend to be intensively performed. Even if the references are distributed, they occur RCCs when they are edited in the same task, such as manual renaming operations.

6.1.3 Unfamiliarity with tools

Many recurrent type name insertions are derived from the developers’ unfamiliarity with content assist. As mentioned in Section 2.1, Eclipse provides a code template function. For example, the code template function can transform the string "cast" into the following code.

```java
type new_name = (type) name;
```
The user can edit the fragment with placeholders. For example, when he/she edits the strings "type", the two appearances of "type" are simultaneously changed. This function has the potential to reduce RCCs drastically.

However, several developers of our dataset did not know the usage of the function. They are not novices. One of the developers (Author 2) has been using Eclipse for over nine years. Roehm et al. pointed out that program comprehension tools built into an IDE are not used in many cases regardless of whether the developers are professional or not [20]. The fact can be applied to also code edit supporting tools. We think that guiding users to reduce inefficient edits and designing tools so that they can be easily started are important.

### 6.2 Design of code completion tools

We assume that RCC-CS is used only when a developer wants to repeat cc-operations and that it is separated from ECC. Thus, there are two code completion tools within the developer's IDE after RCC-CS is installed. This may be problematic in terms of usability.

The developer can select our tool if he/she remembers that he/she inserted the same identifier by code completion previously. Identifiers inserted a long time ago may be forgotten. We think it is not very problematic because most RCCs are performed with short time intervals as shown in Section 4.

To check how much RCC-CS degrades ECC's ranking when inserting not repetitive candidates, we performed an experiment with toy programming. As the result, the degradation is not very crucial so far. It may be because of such small programs often include code fragments conforming Java conventions mentioned in Section 6.1.1. The degradation should be evaluated in future.

It is better to merge the two tools while the total performance is kept well to avoid users' load of switching the tools. Most code completion tools provide a single candidate list. So, we need a ranking algorithm that properly sorts repetitive candidates and the others. If a code completion tool always prioritizes candidates which were previously inserted within a threshold time (e.g., 135 seconds), it ends in neglecting the traditional strategy. To obtain a better merged tool, we must identify exactly when the developer wants to perform RCCs. The merged tool can automatically switch between multiple code completion strategies depending on the current situation. However, our trials to build such a tool unfortunately ended in low performance in predicting RCCs so far. Precise RCC prediction is future work.

### 6.3 Implementation restriction

To simplify tool implementation, the current version of RCC-CS does not discriminate an identifier's type, but ECC does. This sometimes degrades the ECC ranking. Consider the following example.

```java
1: int num = Integer.
2: String str = Integer.
```

In this case, `parseInt("123")` should be entered in line 1, and `toBinaryString(123)` in line 2. `parseInt()` returns an int value. `toBinaryString()` returns a string. However, RCC-CS prioritizes `parseInt()` in line 2 due to the cc-operation in line 1. ECC correctly ranks `toBinaryString()` first in this case.

Although this problem is not a crucial issue in our experiment (Section 5.2), it should be remedied before RCC-CS is used in actual software development.

### 6.4 Threats to validity

Our dataset is derived from the operations of four developers, none of which are business programmers. The characteristics of operation histories are strongly affected by individual developers. However, given the factors of RCCs (Section 6.1), many developers suffer from RCCs regardless of their background.

Our dataset includes several failures of recording developers’ operations. The extracted cc-operation groups do not include such failures since we checked the consistency of operations. However, the failures may prohibit detection of cc-operations.

OperationRecorder cannot record all operations with regard to code completion, such as code completion cancellations. Although we designed our...
experiments to reduce failures, we must create and use a tool that can record such operations to improve experimental validity.

In this work, the target language is Java and the target environment is Eclipse. Other languages and IDEs may differ from Java and Eclipse.

7 Related works

Although we have not yet surveyed all existing IDEs, we are unable to find an IDE that supports a repetitive code completion strategy by default.

An Emacs plug-in called Dynamic Macro [10] can detect repetitive operation sequences and repeat them. The tool does not support code completion and repeats only the last repetition. The vi editor also supports repeating the same operation. These functions are effective but insufficient. As shown in our experiment, many cc-operations repeat previous operations that are not the most recent one. Simultaneous editing is a method to automate repetitive text editing [12]. It requires selecting regions to be edited and does not predict the input candidates.

Robbes and Lanza proposed a benchmark for code completion tools and improved the existing code completion mechanism [19]. They used AST change histories recorded on Squeak IDE, so that code changes of evolving software can be traced. The paper reports that focusing on a current development session and recent changes can make big improvements on code completion. As described in Section 5, our method has a better performance.

Bruch et al. proposed a code completion system that learns from existing code repositories [2]. Their method uses a class and a method as a context of the method invocation and orders code completion candidates based on the context and frequency of method calls within code repositories.

Hou and Pletcher proposed methods to order code completion candidates based on API type hierarchies and API usage frequency, to filter candidates based on user-definable rules, and to group based on API functional roles [6]. Their mechanism does not take recurring cc-operations into account. Nguyen et al. proposed a code completion approach called GraPacc [15]. It constructs a graph-based model of API usage patterns, creates queries from incomplete code, and provides context-sensitive code completion. Yamamoto et al. proposed a tool to improve code completion based on a source code corpus built from existing source code [21]. Because none of these approaches use developers’ operations, they cannot achieve an operation-based context (the location, content, and time of a cc-operation) to improve code completion.

As for code completion for APIs, we can use existing mechanisms with regard to API usage pattern mining [1][22]. Moreover, Han et al. proposed a code completion mechanism based on an abbreviated input [4]. Omar et al. proposed an architecture to facilitate the extension of a code completion mechanism [16]. Mooty et al. extended a code completion mechanism using crowdsourcing to help developers understand and correctly use APIs [13]. These approaches improve existing mechanisms, but do not focus on improving ranking of code completion. Thus, combining these approaches with our method may improve code completion.

We discussed factors of repetitions based on our experiment. Here, we have an additional discussion regarding the factors based on several literatures. Kim et al. reported code clone causes repetitive edits in several cases [8]. Meng et al. proposed systematic edits to perform effective patch application [11]. Such systematic edits by hand often include repetitive operations in our opinion. Since developers continuously write a lump of code in general, repetitive code patterns [5] located in a narrow area tend to cause RCCs.

8 Conclusion

This paper presented several experimental results and discussions in terms of repetitive code completion. Our experiment showed that RCCs occur more frequently when the time interval between the precedent cc-operation and the succedent one is short. Moreover, we implemented RCC-CS and presented an evaluation. The result shows that the tool outperforms an existing method. Based on the results, we concluded our statement mentioned in Section 1 is accurate. In the future, we plan to explore collaborations between a repetitive strategy and other strategies, while preserving the usability and performance for actual software development. Additionally, we will clarify the factors that cause
RCCs.

Acknowledgments

This work was supported by MEXT/JSPS KAKENHI Grant Number 26730042, 24500050, and 24700036. We also thank the anonymous reviewers for the many useful suggestions.

References


corresponds to the state before a developer starts any input regarding the cc-operation. The final state corresponds to the state right after a candidate is inserted. The with-completion-candidates indicates the state in which a code completion list has popped up. The without-completion-candidates indicates the opposite of it. The descriptions of transition events are attached to each arrow between event nodes. A bracketed string following an event description indicates the operation(s) OperationRecorder records when the event occurs. This notation complies with the former notation rule. For example, starting from the initial state, if a developer typed a character and a candidate list popped up without an explicit operation such as hitting Ctrl+Space keys, and he/she subsequently selected a candidate from the list, the recorded operation pattern is represented as \([\text{NC}]\).

Here we must pay attention to the transition of the content of normal operations. Any normal operation may change its content until the next operation is generated. We call such a changeable operation an open operation. For example, when a developer types \(a\), a normal operation whose inserted string is \(a\) is generated. Subsequently, if he/she types \(b\), the inserted string changes into \(ab\). Conversely, a non-changeable operation is called a closed operation. In Figure 6, the description [close] means to change an open normal operation located at the tail of the operation fragment into a closed one. Even if the last operation is open, the operation closes when a new normal operation is added due to switching insertion and deletion. The description “ESC” means pressing the escape key to close the candidate list. The description \([\phi]\) indicates that the operation does not affect the operation fragment. The description “content assist” indicates that the developer explicitly invoked content assist. If there is only a single candidate, the candidate is directly inserted without showing a candidate list.

### B Code completion examples

We describe three examples of code completion and their corresponding state transitions in Figure 6.

**Example 1: Inserting a single identifier**

Table 4 shows edit events, transitions of source code and states, and the recorded operation history in this example. A developer types \(st\). Then, he/she launches code completion by hitting Ctrl+Space keys. Finally, he/she selects \(str\) from the code completion candidate list. In the state column, ×, √, and T indicate “without-completion-candidates”, “with-completion-candidates”, and the final state, respectively. In the operation history column, we use the notion explained in Section 3.3.3. It indicates the operation history fragment that OperationRecorder keeps internally after each event. Additionally, the parenthesized string in the operation history column indicates details of the last operation in the operation history fragment. In an actual operation history, the file name and the offset where the operation is performed are recorded. However, they are omitted here to simplify the explanation. In the examples, the operations’ edit places are not distributed.

The operation history for event 1 in Table 4 represents an operation history fragment including a single normal operation that inserts \(st\). The operation history for event 2 represents the inserted string by the normal operation is changed into \(st\) since the normal operation is still open. By invoking content assist (event 3), the operation is closed.

<table>
<thead>
<tr>
<th>Edit event</th>
<th>Code</th>
<th>State</th>
<th>Operation history</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Type ‘s’</td>
<td>s</td>
<td>×</td>
<td>[N] (ins &quot;s&quot;)</td>
</tr>
<tr>
<td>2 Type ‘t’</td>
<td>st</td>
<td>×</td>
<td>[N] (ins &quot;st&quot;)</td>
</tr>
<tr>
<td>3 Content assist</td>
<td>st</td>
<td>√</td>
<td>[NM]</td>
</tr>
<tr>
<td>(Non-uniq.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Cand. Selection</td>
<td>str</td>
<td>T</td>
<td>[NMC] (del &quot;st&quot;, ins &quot;str&quot;)</td>
</tr>
</tbody>
</table>
(its inserted string is fixed) and a new menu operation is added. Finally, by selecting the candidate, a compound operation is added. The operation replaces the prefix string (i.e., "st": the string that the developer manually typed) with the whole identifier ("str").

**Example 2: Implicit code completion**

Table 5 shows an overview of this example. A developer types "s" that is a variable whose type is `String`. Then, he/she types ".". Because of this insertion, code completion is invoked automatically. So, the state is transferred to "with-completion-candidates". Then, he/she types "com" to restrict the candidates. Finally, he/she selects "compareTo()" from the candidate list. ECC inserts the selected candidate with the formal argument ("anotherString") in the parentheses.

As shown in Table 5, the final operation history fragment is [NC] in this case. In our dataset, such a pattern is comparatively rare. That means the developers explicitly invoked code completion in most cases.

**Example 3: Mistyped prefix**

Table 6 shows an overview of this example. A developer types "s", a `String` variable. Then, he/she types ".". Because of this insertion, code completion is invoked automatically, as mentioned in Example 2. Then, he/she types "conx". When "x" is inserted, the code completion candidate list disappears since there are no candidates starting "conx". Here, "x" is a typo. So, he/she deletes it and types "c". The candidate list does not reappear since the trigger of its appearance is inserting ".". Then, he/she invokes code completion explicitly. In this case, there is only a single candidate starting "conc". So, the candidate ("concat(arg0)") is automatically inserted. In this case, multiple normal operations are recorded before [M] since switching insertion and deletion separates normal operations. As the result, the last normal operation before the compound operation contains "c" as its inserted string, while the compound operation contains "conc" as its deleted string. As explained the last paragraph of Section 3.3.3, we check whether those strings are the same. So, the operations are not determined as a cc-operation group. In this paper, the investigation target is limited to cc-operations without typos.
Takayuki Omori
is an assistant professor at the Department of Computer Science, Ritsumeikan University. He obtained his Ph.D. in Engineering from Ritsumeikan University in 2008. From September 2013 to March 2014, he was a visiting assistant professor at the University of British Columbia. His current research interests include software evolution, and fine-grained code changes in software development.

Hiroaki Kuwabara
is an assistant professor at the Department of Computer Science, Ritsumeikan University. He received his Ph.D. degree in Information Science from Nagoya University in 2007. His current research interests include software engineering, program analysis, formal methods, type systems, and communicating process models.

Katsuhisa Maruyama
is a professor at the Department of Computer Science, Ritsumeikan University. He received Ph.D. (1999) in Information and Computer Science from Waseda University. He has worked for NTT (1993-1999) and NTT Communications Corporation (1999-2000). He is a member of the IEEE Computer Society, and ACM. He was a visiting researcher at the Institute for Software Research (ISR) of University of California, Irvine (UCI) from September 2003 through September 2004.