An Empirical Study on Change Recommendation

Manishankar Mondal
Department of Computer Science
University of Saskatchewan, Canada
mam815@mail.usask.ca

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan, Canada
chanchal.roy@usask.ca

Kevin A. Schneider
Department of Computer Science
University of Saskatchewan, Canada
kevin.schneider@usask.ca

ABSTRACT

Recommending changes to programmers by exploiting their repetition tendencies during system evolution has been investigated by a number of studies. In our research we perform a change type (additions, deletions, and modifications) based analysis of the efficiency of change recommendation. We also investigate the programmer sensitivity of the repeated changes (i.e., the extent the same changes are repeated by the same programmers) of different change types. The existing studies did not perform such investigations. However, these investigations can be important for efficient ranking (i.e., prioritizing) and filtering of recommendations. According to our investigation on thousands of commits of five diverse subject systems we observe that modifications have a very low tendency (around 1.3%) of being repeated. We should primarily focus on recommending additions, and deletions. More importantly, overall 71% of the repeated changes are programmer sensitive. We believe that a change recommendation system that prioritizes recommendations considering programmer sensitivity can help programmers reuse previous changes in a time-efficient manner.

1. INTRODUCTION

Software systems will undergo changes during evolution. However, making changes to a software system is often challenging [24,25]. First, manual implementation of the changes is often time consuming. Second, program comprehension may take additional time for even experienced programmers for a reasonably large system before making the changes. Third, a complete understanding of the change requirements is required for making the changes, and fourth, any inaccurate changes might have severe negative impacts on the maintenance and evolution of a software system. In such a situation, it would be beneficial if we could automatically provide change suggestions to a programmer when she is going to make some changes to a code snippet.

A number of studies [32, 37] have investigated the possibility of change recommendations on the basis of the past evolution history of a software system. The main idea behind these existing approaches is simple. Let us assume that a programmer is currently working on a code snippet $CS_{current}$. She wants to make some changes to it. However, it is possible that a similar code snippet $CS_{previous}$ was previously modified in the same way she is now thinking, and in this case, the change that occurred to $CS_{previous}$ can be recommended to the programmer for making a similar change to the target code snippet $CS_{current}$.

Although the existing change recommendation techniques [32, 37] are good at providing useful recommendations during software development and maintenance, we conduct the following two investigations in order to improve the recommendation efficiencies of the existing techniques. None of the existing studies on change recommendation performed such investigations.

(1) Investigating the efficiency of change recommendation considering different change types. An investigation on the comparative efficiency of recommending different types of changes (i.e., additions, deletions, and modifications) is important. If it is observed that the recommendation efficiency regarding a particular change type is very low or negligible compared to the other change types, then we might decide to filter out the recommendations of that particular change type. Such discarding of recommendations can reduce the number of false positives, and thus, can minimize the cognitive burden of the programmers in selecting the most suitable one from too many recommendations. However, the previous studies did not focus on such an investigation. Nguyen et al. [32] investigated repetition tendencies of changes in different program constructs (such as If, For, While). They did not investigate the three change types: additions, deletions, and modifications. Thus, from their investigation we cannot infer the comparative efficiencies in recommending these three types of changes.

(2) Investigating programmer sensitivity of the repeated changes of different change types. If most of the repeated changes are programmer sensitive (i.e., if the same repeated changes are generally made by the same programmers), then we can exploit this fact in order to rank the recommendations for changing a target code fragment. If a particular programmer is going to change a particular target code snippet, then we can prioritize change suggestions inferred from those previous changes where that particular programmer changed similar code snippets (i.e., similar to the target snippet). Such prioritizations can also minimize the cognitive burden of the programmers in understanding and selecting the most suitable change suggestion from the whole list of suggestions. However, the existing studies did not investigate this matter.

Focusing on the above two issues we perform a change type (additions, deletions, and modifications) based analysis on the recommendation efficiency and programmer sensitiveness of change recommendations. First, we implement a light-weight change suggestion mechanism that provides suggestions by analyzing the past evolution history of a software system. We implement our mechanism using UNIX *diff* and considering the exact similarity of code fragments. Then, we apply our implementation on thousands of commits of five diverse subject systems for performing our investigations. We answer the following two research questions from our investigations.

- **RQ 1:** Which type(s) of changes can be suggested more effectively compared to others?
- **RQ 2:** Are the repeated changes generally programmer sensitive?

According to our experimental results we can state that:

(1) Modifications have an almost negligible tendency (1.3%) of being repeated. We can possibly exclude modifications from consideration while providing change recommendations. We should primarily focus on recommending additions and deletions.

(2) We find that around 71% of the repeated changes are programmer sensitive. In other words, the same repeated changes are usually made by the same programmers during evolution. Thus, while ranking change suggestions for a particular programmer, we should prioritize those suggestions that are inferred from the previous changes made by that particular programmer.

The rest of the paper is organized as follows. Section 2 elaborates on the procedure that we follow for providing automatic change suggestions, Section 3 discusses the experimental steps, Section 4 answers the research questions by presenting and analyzing the experimental results, Section 5 discusses the related work, Section 6 mentions the possible threats to validity, and finally, Section 7 concludes the paper by mentioning possible future work.

2. CHANGE SUGGESTION MECHANISM

For the purpose of our empirical study we implement a light-weight change suggestion mechanism using UNIX *diff* and considering the exact similarity of the code fragments. A number of change suggestion techniques [32, 37] already exist. These techniques are based on the syntactic similarity of code fragments. As a result, the suggestions they provide are sometimes templates (i.e., suggestions with dummy variable names and literal values) and cannot be readily applied to the target code fragment [37]. Sometimes automatic program generation tools [13, 19, 22, 23] can help us adapt a suggested template to the relevant change context. However, an accurate adaptation can often require interactions from the programmers. We wanted to avoid template suggestions in our study in order to find the precise answers to our research questions. For this reason we implemented our change suggestion technique considering exact similarity of code fragments. The suggestions that we provide can always be readily applied to the target fragment. In the following subsections we describe our change suggestion mechanism.

2.1 Change Detection

We detect the changes using the UNIX *diff* command. A number of change detection tools [3, 18] are currently available. However, we use *diff* in our experiment because it is light-weight. Let us assume that a number of files were changed in a particular commit operation. We consider a particular file F from these changed files. We obtain two instances, F_{before} and F_{after} of the file F where F_{before} is the file in the revision before the commit operation and F_{after} is the file in the revision created after the commit. We determine the *diff* of these two files. We get three types of changes from *diff* output. These are: *addition*, *deletion*, and *modification*. We provide a simple example in Fig. 1 for demonstrating these change types.

Fig. 1 shows two files, *File 1* and *File 2*, and the output after applying *diff* on these two files. We show the line numbers for the lines in the files *File 1* and *File 2*. These line numbers help us understand the *diff* output. From the *diff* output we realize that there are three types of changes: *addition*, *deletion*, and *modification*. *diff* indicates these change-types by 'a' (for addition), 'd' (for deletion), and 'c' (for modification) respectively. We should note that the letter 'c' in the first line '3,4c3,4' of *diff* output indicates change or modification of existing lines. In the rest of the paper we will use the term *modification* for this.

We now analyze the *diff* output. The first line '3,4c3,4' in the output indicates that the lines 3 and 4 of *File 1* were *modified*. The corresponding lines in *File 2* after the *modification* are also 3 and 4. The lines before and after the modification are shown afterwards. The line '6,7d5' indicates that the lines 6 and 7 in *File 1* were deleted, and this deletion happened just after the 5th line in *File 2*. The deleted lines are also shown in the *diff* output. The line '9a8,9' indicates that two lines were added after line 9 in *File 1*. These two lines correspond to lines 8 and 9 in *File 2*. Finally, line '10a11' indicates that a new line was added after line 10 in *File 1*. This newly added line corresponds to line 11 in *File 2*.

2.2 Change Suggestion

Let us assume that a particular programmer wants to make some changes to the code-base of a particular software system. She at first identifies the place where to implement the change and finally, clicks a line L from where she wants to begin the change. She can do the following three types of changes as defined in Section 2.1.

(1) Addition of one or more lines after L.

(2) Deletion of one or more consecutive lines beginning with the particular line L.

(3) Modification or changing of a code snippet consisting of one or more consecutive lines beginning with L.

In such a situation, we provide a list of change suggestions to the programmer so that she can choose a particular suggestion and can apply the corresponding change. As there are three types of changes, we also provide three types of change suggestions to the programmer. In the following subsections we describe the mechanisms that we follow for providing change suggestions.

2.2.1 Suggesting Addition.

We have already mentioned that the programmer clicks the line L she wants to start changing from. In this subsection we discuss how we provide automatic suggestions for

```
 1:  A
 2:  B
 3:  this line will be modified 1
 4:  this line will be modified 2
 5:  C
 6:  this line will be deleted 1
 7:  this line will be deleted 2
 8:  D
 9:  E
10:  F
11:  G
12:  H
```

(a) File 1

```
 1:  A
 2:  B
 3:  this line is modified 1
 4:  this line is modified 2
 5:  C
 6:  D
 7:  E
 8:  a new line is added 1
 9:  a new line is added 2
10:  F
11:  a new line is added 3
12:  G
13:  H
```

(b) File 2

```
3,4c3,4
< this line will be modified 1
< this line will be modified 2
—
> this line is modified 1
> this line is modified 2
6,7d5
< this line will be deleted 1
< this line will be deleted 2
9a8,9
> a new line is added 1
> a new line is added 2
10a11
> a new line is added 3
```

(c) *diff* between File 1 and File 2

Figure 1: An example of *diff* output showing the three types of change: *addition*, *deletion*, and *modification*.

making additions after line L. We first assume that we have a database containing all the additions made during the past evolution of the software system. Each occurrence of addition is stored with the following information: (1) the line after which the addition was made, and (2) the lines that were added. We get these from the *diff* output as described in Section 2.1. We automatically examine each of the additions recorded in the database and identify those ones where the line after which the addition was made is similar (similarity is defined later in this section) to L. These identified occurrences of addition are the suggested additions for the programmer.

2.2.2 Suggesting Modification.

In this section, we discuss our mechanism for suggesting modifications for line L (i.e., the line from which the programmer wants to start changing) or for a set of consecutive lines starting from line L.

We assume that we have a database of all the modifications that occurred during the past evolution of the system. A particular occurrence of a modification stored in the database contains the following two pieces of information.

(1) The prior code snippet. This is the code snippet (i.e., a code block containing one or more consecutive lines) prior to the occurrence of the modification.

(2) The posterior code snippet. This is the code snippet that was obtained after the occurrence of the modification to the prior code snippet.

We obtain this information from the *diff* output as discussed in Section 2.1. We now automatically examine the modifications stored in the database and identify each modification occurrence where the prior code snippet is similar to a code snippet of the same length beginning from line L in the code-base the programmer is now working on. We consider these identified modifications as suggestions for the programmer.

2.2.3 Suggesting Deletion.

Lastly, we discuss our mechanism for suggesting deletion of the line L or of a set of consecutive lines starting from L. As before we again assume that we have a database containing all the occurrences of deletions from the evolution history. Each occurrence of deletion stored in the database contains a code snippet (i.e., one or more than one consecutive lines) that was deleted. We examine the deletions stored in the database and select each deletion occurrence where the deleted code snippet is similar to a code snippet of the same length starting from the line L that the programmer clicked. These selected occurrences of deletion are considered the suggested deletions for the programmer.

2.2.4 Storing the changes that occurred during the past evolution of the software system.

We examine each of the commit operations that occurred to the code-base of the software system. Let us consider a particular commit C. While examining the commit operation C we determine which source code files were modified in this commit. For each of these files we collect two snap-shots: **(1)** the snapshot of the file in the revision on which the commit C was applied, and **(2)** the snap-shot of the file in the revision that was created after the application of the commit operation C. We apply *diff* on these two snap-shots. From the *diff* output we identify all the changes (additions, deletions, or modifications) that occurred to the file in commit C. In this way we identify all the changes that occurred to all the files modified in commit C. We store these changes in the database as mentioned in Section 2.2.1, 2.2.2, and 2.2.3. Each occurrence of a change is stored in the database along with the commit-ID. We do this for the purpose of our empirical study.

2.2.5 Detection of Similarity between Code Snippets.

We have mentioned that we detect similarity between code snippets. For this purpose we use an adapted version of NiCad [6, 40] technology where we can detect the similarity of two given code snippets of any granularity (i.e., block or method). Let us assume that we have two code snippets, s_1 and s_2. We consider that these two code snippets are similar if NiCad detects them as exact clones of each other. We use NiCad in our study, because NiCad is capable of detecting clones with high precision and recall [38–40].

3. EXPERIMENTAL STEPS

We perform our investigation on the five subject systems listed in Table 1. We see that the starting revisions for jEdit

Table 1: Subject Systems

Sys.	Lang.	Domains	LLR	SRev	ERev
Camellia	C	Image Processing Library	88,033	1	170
jEdit	Java	Text Editor	191,804	3791	4000
Freecol	Java	Game	91,626	1000	1950
Carol	Java	Game	25,091	1	1700
Jabref	Java	Reference Management	45,515	1	1545

SRev = Starting Revision ERev = End Revision
LLR = LOC in the Last Revision

and Freecol are respectively 3791 and 1000. The reason behind this is that the SVN repository location from which we obtained the code-base did not contain the code for the missing revisions. This might happen as a result of changing the directory structure just after the revisions 3790, and 999 in case of jEdit and Freecol respectively. We perform the following preliminary steps for each of the systems:

(1) Download each revision (as noted in Table 1) of the systems from the on-line SVN repository [35],

(2) Preprocessing of the source code files in each revision by removing comments, blank lines, and indentations,

(3) Detect changes (additions, deletions, and modifications) between the corresponding preprocessed source code files of every two consecutive revisions by applying UNIX *diff*,

(4) Store the changes in the database with the respective commit-IDs. We store each of the three types (additions, deletions, and modifications) of changes in the database. The details of which pieces of information are stored for each type of change is described in Section 2.

(5) Determine which developer made which changes over the evolution by using the *SVN log* command. This command not only retrieves the log message for each commit operation but also shows which developer was responsible for which commit. We retrieve this information for answering *RQ 2*.

We perform preprocessing of the source code files because we wanted to investigate change recommendations for the source code lines only (i.e., not for the comments). After applying the above experimental steps on the subject systems we get the experimental results. We then analyze these experimental results for answering the research questions.

4. EXPERIMENTAL RESULTS

In this section we answer the two research questions mentioned in the introduction by presenting and analyzing our experimental results. Before answering research questions, it is important to analyze and report the accuracy of our implemented change suggestion mechanism. In the following paragraphs we describe the procedure of evaluating our change suggestion mechanism.

Evaluating our Change Suggestion Mechanism

Our evaluation procedure is a variation of the *n-fold cross-validation* technique in which we sequentially examine the commit operations starting from the very first revision mentioned in Table 1. While examining a particular commit operation C, we determine whether we can suggest the changes that occurred in this commit (i.e., C) by analyzing the changes that occurred in the previous commit operations (i.e., the commits from 1 to $C-1$).

Let us consider that we are currently examining the commit operation C. A change c_{current} occurred to a particular code snippet s_{current} in this commit. If we can suggest this change (i.e., c_{current}) for this code snippet (i.e., s_{current}) by analyzing the previous commits, 1 to $C-1$, then we can say that we can provide accurate change suggestions using our approach. Thus, in this context our goal is to determine whether and to what extent we can suggest the change c_{current} for the code snippet s_{current} by analyzing the previous evolution history. The following three cases can happen in such a situation.

Case 1. We can provide change suggestions that include the accurate change (c_{current}). We automatically mine the evolution history consisting of the commits 1 to $C-1$ and find that one or more similar (similarity is defined in Section 2.2.5) code snippets were previously changed. We extract these previous changes and provide these as suggestions for changing the current code snippet $s_{current}$. The suggestions include the particular change c_{current}. Thus, in this case the change suggestions that we provide contain the accurate change to be implemented. We can easily understand that an accurate change suggestion is possible only in the case that there are repeated changes.

Case 2. We can provide change suggestions, however these suggestions do not include the accurate change (c_{current}). We mine the evolution history and find that one or more similar code snippets were previously changed. We provide these changes as suggestions for changing the current code snippet s_{current}. However, the suggestions do not include the particular change c_{current}. Thus, in this case we cannot provide an accurate change suggestion.

Case 3. We cannot provide any change suggestion. In this case we find that no similar code snippets were previously changed. Thus, we cannot provide any suggestions for changing the particular code snippet s_{current}.

For the first and second cases above, we need to check whether a change appearing in the suggested list is similar to the particular change c_{current} that occurred to the code snippet s_{current} in commit C. We check the similarity between two changes, c_{current} and c_{previous} (occurred in a previous commit C_{previous} where $C_{\text{previous}} < C$), in the following way.

(1) Let us consider that each of c_{current} and c_{previous} is an *addition*. These two changes are similar if the following two conditions hold: (i) the two lines after which the additions were made in the two cases are similar, and also, (ii) the two code snippets (containing one or more lines) added in these two cases are similar. We detect exact textual similarity following the procedure described in Section 2.2.5.

(2) Let us consider that c_{current} and c_{previous} are *deletions*. These two changes are similar if the two code snippets that were deleted in the two cases are similar.

(3) Let us consider that c_{current} and c_{previous} are *modifications*. These two changes are similar if the following two conditions hold: (i) the two code snippets that were modified in the two cases are similar, and also, (ii) the two code snippets obtained after the modifications in the two cases are similar.

We automatically examine each of the changes that occurred in a particular commit C and determine whether we can accurately suggest these changes by analyzing the changes that occurred in the previous commits 1 to $C-1$.

Table 2: Statistics Regarding Change Suggestion During the Evolution of our Subject Systems

	Camellia	jEdit	Freecol	Carol	Jabref
TC	3115	6235	16308	8241	14901
TS	445	1187	4969	1573	2630
RC	65	200	753	370	375

TC = Total number of changes during system evolution.
TS = Total number of cases where we can provide one or more change suggestions.
RC = Total number of repeated changes during evolution. This is the total number of cases where our suggestions contain the actual change to be implemented.

By examining all the changes that occurred in all the commit operations during the evolution of a software system we determine the followings and record these in Table 2.

(1) TC (in Table 2) is the total number of changes that occurred during these commits. This number is the total number of occurrences of *Case 1*, *Case 2*, and *Case 3* (described earlier in this subsection) during evolution.

(2) TS is the total number of cases for each of which we could provide one or more change suggestions (whether accurate or not). This number is the number of occurrences of *Case 1* and *Case 2* during evolution.

(3) RC is the total number of cases for each of which we could provide an accurate change suggestion. This number is the number of occurrences of *Case 1* during evolution. An accurate change suggestion is possible only in the case of change repetition. Thus, the total number of accurate change suggestions is the total number of repeated changes during evolution. We define a repeated change in the following way.

Repeated Change. If a particular change c occurred in a particular commit operation C is similar to one or more changes that occurred in the past commits 1 to $C - 1$, then we say that the change c in commit C is a repeated change. We have already described the similarity of two changes.

We also determine the following three percentages considering the values recorded in Table 2.

(1) ***The percentage of cases where we can provide one or more change suggestions.*** This percentage (i.e., TS × 100 / TC from Table 2) is shown in Fig. 2. We see that for each of the subject systems we can provide change suggestions for a considerable percentage of cases. The overall value (i.e., considering all subject systems) of this percentage is 22.14%. In the case of our subject system Freecol, this percentage (i.e., around 30.47%) is the highest.

(2) ***The percentage of cases where the suggestions that we provide contain the actual change (i.e., the accurate suggestion) to be implemented.*** This percentage (i.e., RC × 100 / TS) is shown in Fig. 3 and it indicates how often the change suggestions that we provide can contain the actual change to be implemented. In other words, this percentage determines the precision of our approach in providing change suggestions. According to Fig. 3, the overall probability (in percentage) that the change suggestions that we provide will contain the accurate change to be implemented is around 16%. This probability is the highest (23%) for Carol. In Nguyen et al.'s [32] study this percentage was

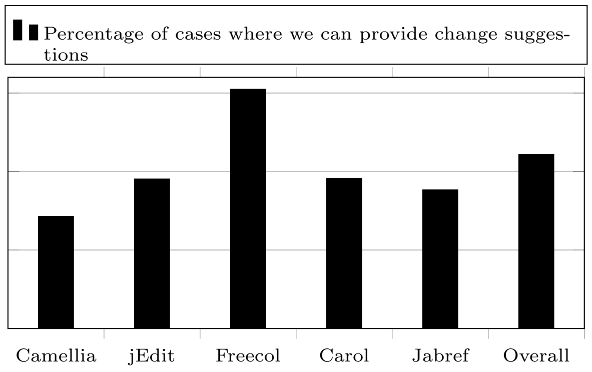

Figure 2: Percentage of cases where we can provide one or more change suggestions.

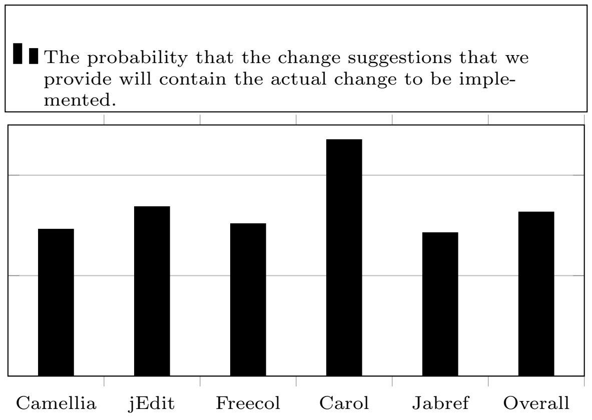

Figure 3: The probability (in percentage) that the change suggestions that we provide will contain the actual change to be implemented.

around 30% which is larger than our overall percentage. The reason is that they considered syntactic similarity in their approach. They report their accuracy in suggesting change templates (i.e., suggestions with dummy variable names and literal values) rather than suggesting the actual change to be implemented [32,37]. However, in our research we wanted to exclude the template suggestions in order to find precise answers to our research questions. Thus, we limited ourselves to exact similarity, and find the accuracy of our mechanism in suggesting the actual change to be implemented.

(3) ***The percentage of repeated changes during the whole period of evolution.*** This percentage (i.e., RC × 100 / TC) is shown in Fig. 4. The figure implies that the percentage of repeated changes is considerable although very low for some subject systems (such as Camellia). Such a finding complies with the findings from the previous studies [32,37]. This percentage is around 5% for our subject system Freecol. The overall value of this percentage is 3.62%.

From our above discussion and analysis regarding the evaluation of our implemented change suggestion mechanism we can state that our implemented mechanism can provide actual change suggestions (the change suggestions that can be readily applied to the target code snippets) with a precision of up to 23% which is reasonable with respect to our considerations and to the

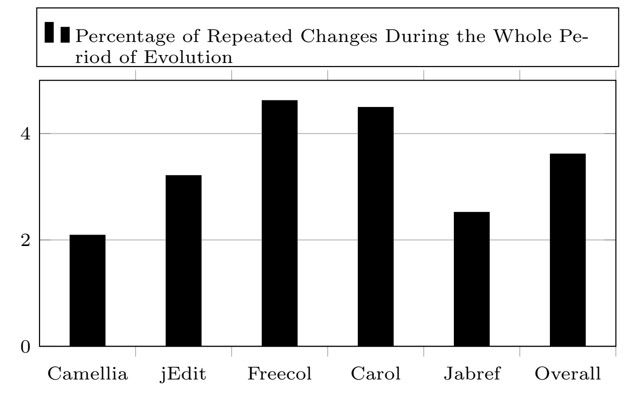

Figure 4: The Percentage of Repeated Changes During the Whole Period of Evolution.

results reported by the previous studies.

We will now answer the two research questions (i.e., mentioned in the introduction) through experiments performed using our implemented change suggestion mechanism.

RQ 1: Which type of changes can be suggested more effectively compared to the others?

If the changes of a particular type (additions, deletions, or modifications) are repeated more frequently than the changes of the other change-types, then we can say that the changes of that particular type can be suggested more effectively compared to the other change-types. Thus, for answering this research question (i.e., *RQ 1*) we compare the likeliness of the repetition of the changes of different change-types. Nguyen et al. [32] compared change repetitions in different language constructs. However, we compare change repetition considering the three change types: addition, modification, and deletion. Such an investigation is important, because the findings from this investigation can provide us with helpful insights for developing a change suggestion plug-in. If it is observed that the likeliness of the repetitions of the changes of a particular change-type is very low or negligible, then we can exclude the changes of that particular type from consideration while developing the plug-in. Such an exclusion can assist in faster execution of the plug-in.

Methodology. Using our implemented change suggestion tool we examine the changes that occurred to each of the commit operations sequentially starting from the very first one. While examining the changes in a particular commit operation C, we determine which of these changes we can accurately suggest by analyzing the previous commits (1 to $C-1$). A change $c_{current}$ in the current commit C can only be accurately suggested if it occurred in at least one of the previous commits, that is, if $c_{current}$ is a repeated change. We determine which of the changes occurring in commit C are repeated changes. We separate these repeated changes into three disjoint sets:

- The set of repeated additions,
- The set of repeated deletions, and
- The set of repeated modifications

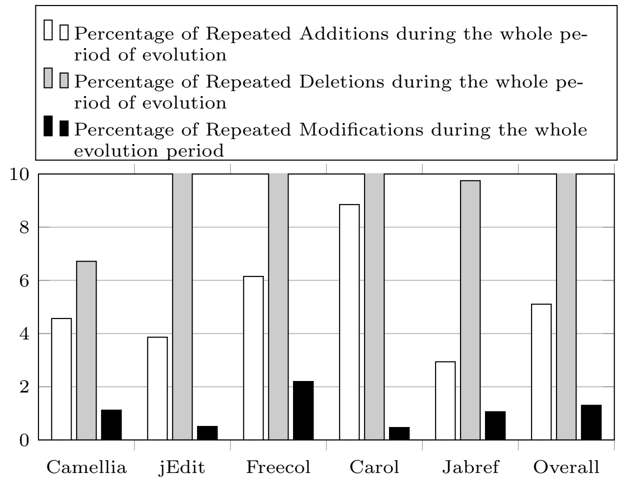

Figure 5: Comparison of the tendency of repetition of different types of changes.

In this way we examine each of the commit operations occurred during the evolution of a subject system and identify the repeated addition, deletions, as well as modifications. We determine the likeliness of the repetition of additions in the following way.

$$LRA = \frac{100 \times \sum_{for\ all\ commits} |SRA|}{\sum_{for\ all\ commits} |SA|} \quad (1)$$

Here, LRA is the likeliness of the repetition of additions during the evolution of a particular subject system. SA is the set of all additions in a particular commit operation, and SRA is the set of all repeated additions in that commit operation. In the same way we also determine the likeliness of the repetition of deletions and modifications. From the equation we see that we determine the likeliness as a percentage. We plot three percentages (i.e., corresponding to the three change types) for each of the subject systems in the graph of Fig. 5.

From the graph (Fig. 5) we understand that modifications have the lowest tendency of being repeated among the three change types. For two subject systems, jEdit and Carol, this tendency is even smaller than one. The repetition tendencies of additions and deletions are much higher than that of modifications for each of the subject systems. Also, deletions exhibit the highest tendency of repetitions. The graph shows the overall repetition tendencies of the three types of changes. We see that while modifications have the lowest overall tendency of repetition, the overall tendency of deletions is the highest.

Statistical Significance Test. We wanted to investigate whether the repetition tendencies of additions and deletions are significantly higher compared to the repetition tendency of modifications. We perform Mann-Whitney-Wilcoxon (MWW) tests [21] for this purpose. Using this test we determine whether the five repetition tendencies of additions from five subject systems are significantly different than the five repetition tendencies of modifications from these systems. We should note that MWW test is a non-parametric test and it does not require the samples to be normally distributed [33]. This test can be applied to both large and small sample sizes [20]. We perform our test con-

Table 3: The number of authors involved in the commits

	Camellia	jEdit	Freecol	Carol	Jabref
CR	1 - 170	3791 - 4000	1000 - 1950	1 - 1700	1 - 1545
CA	1	3	17	25	18

CR = Commit Range.

CA = Count of Authors involved in the commits.

sidering a significance level of 5%. According to our test, the repetition tendencies of additions are significantly different than the repetition tendencies of modifications with a p-value of 0.0128 (for two-tailed test case) which is smaller than 0.05, and with a large effect size [10] of 0.79. The effect size calculation procedure for MWW test is available on-line [11]. From our test results we can say that the repetition tendency of additions is significantly higher than the repetition tendency of modifications. From a similar test we also find that the deletions have a significantly higher repetition tendency than modifications.

Answer to RQ 1. According to our experimental results and analysis, *deletions can possibly be suggested more effectively compared to the other two types of changes. The repetition tendency of additions is also considerable for most of the subject systems. However, the likeliness of the repetition of modifications is the lowest among the three change types. Our statistical significance tests show that modifications exhibit a significantly smaller tendency of repetition compared to the repetition tendencies of additions and deletions.*

Our experimental results imply that while developing a change suggestion tool or plug-in we should primarily focus on repeated additions and deletions. The overall percentage of repeated modifications is very low (around 1.3%). Such a finding implies that we can possibly ignore modifications while providing change suggestions. However, in order to generalize this finding we need to investigate more subject systems of different programming languages. We plan to do this as a future work.

RQ 2: Are the repeated changes generally programmer sensitive?

If it is observed that most of the repeated changes are programmer sensitive (i.e., the same repeated changes are generally made by the same programmers), then we can use this fact to build a change suggestion tool that can provide programmer sensitive change suggestions. Refining or ranking of the change suggestions on the basis of the programmer who is currently working on the code-base, can not only assist in faster execution of the change suggestion tool but also can assist programmers select change suggestions from a more precise list.

Methodology. For determining which programmer is responsible for which changes we retrieve and analyze the SVN commit log by applying the *SVN log* command. The log for a particular commit operation contains information about which programmer made the changes in that commit. In Table 3 we show the numbers of authors involved in the commits in the commit-ranges (as recorded in Table 1) of

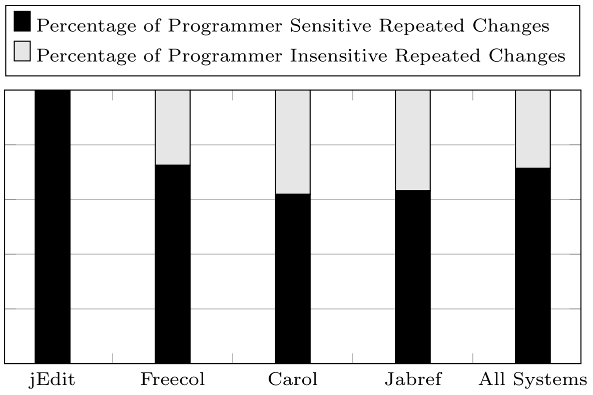

Figure 6: Percentage of Programmer Sensitive Repeated Changes During Software Evolution.

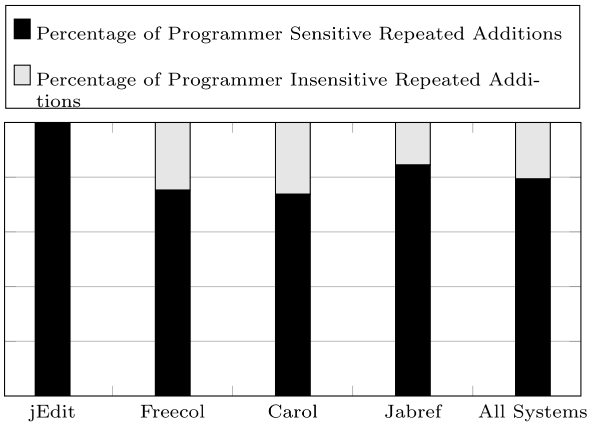

Figure 7: Percentage of Programmer Sensitive Repeated Additions During Software Evolution.

the subject systems. We see that only one author committed during our investigated commit range (1 to 170 as mentioned in Table 1) of Camellia. For this reason we exclude Camellia from consideration for avoiding possible bias in determining the programmer sensitiveness of repeated changes. We perform our investigation by applying our implemented change suggestion mechanism in the following way.

Let us consider that we are now examining the changes in a particular commit C. The programmer who performed this commit operation is P_C. We identify all the repeated changes in this commit operation using our implemented change suggestion mechanism. Suppose, c_{repeated} is a particular repeated change. We determine the older commits where c_{repeated} occurred before. For each of these older commits we determine the corresponding programmer who performed the commit. We check whether the same programmer (i.e., P_C) who performed the commit operation C also performed any of these older commits. If this is true, then we mark the repeated change c_{repeated} as a *programmer sensitive repeated change*. Otherwise, we mark c_{repeated} as a *programmer insensitive repeated change*. By examining the repeated changes in a particular commit C we determine the following measures.

(1) The number of all repeated changes.

(2) The number of *programmer sensitive repeated changes*.

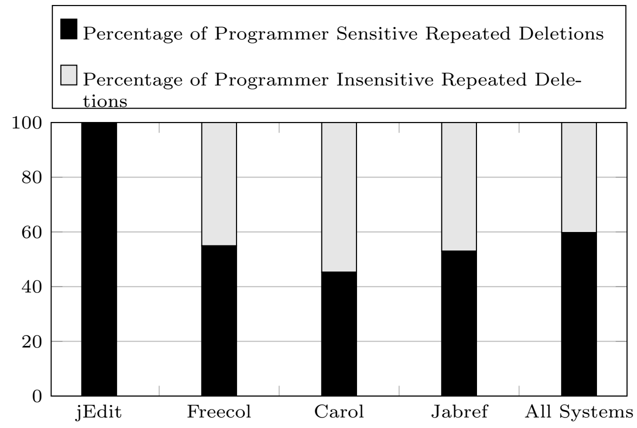

Figure 8: Percentage of Programmer Sensitive Repeated Deletions During Software Evolution.

Considering all repeated changes in all the commit operations we determine the overall percentage of programmer sensitive repeated changes using the following equation.

$$\% \ of \ PSRC = \frac{100 \times \sum_{for \ all \ commits} PSRC}{\sum_{for \ all \ commits} RC} \quad (2)$$

In the above equation, RC denotes the number of *repeated changes* in a particular commit. $PSRC$ is the number of programmer sensitive repeated changes in that particular commit. We also determine the overall percentage of *programmer insensitive repeated changes* using Eq. 3.

$$\% \ of \ PIRC = 100 - (\% \ of \ PSRC) \quad (3)$$

We plot these two percentages in a stacked bar graph shown in Fig. 6. From the graph we see that for each of the subject systems the percentage of programmer sensitive repeated changes is much higher than the percentage of programmer insensitive repeated changes. In case of our largest system jEdit, all of the repeated changes were programmer sensitive. The overall percentage (considering all systems) of programmer sensitive repeated changes is around 71%.

We also wanted to investigate how the programmer sensitivity of the repeated changes differs across the three change types (additions, deletions, and modifications). Considering each type of change separately, we determine the percentages of programmer sensitive as well as programmer insensitive repeated changes and plot these percentages in a graph similar to the graph in Fig. 6. For three change types - addition, deletion, and modification we show three such graphs in Fig. 7, 8, and 9 respectively.

The graphs in Fig. 7 and 9 imply that the repeated additions and modifications are mostly programmer sensitive. The overall percentage of repeated additions as well as repeated modifications is around 80%. From Fig. 8 we also see that the overall percentage of programmer sensitive repeated deletions is higher than the overall percentage of programmer insensitive repeated deletions. We see that in case of only one system, Carol, the repeated deletions are mostly (54.71% of the repeated deletions) programmer insensitive.

Answer to RQ 2. According to our analysis and discussion for this research question we can state that *most of the repeated changes (i.e., around 71% of the repeated changes) that occurred during the evolution of a software system are programmer sensitive.* In other words, *the same repeated changes are usually made by the same programmers.*

Our investigation on different change types implies that the repeated changes of each of the change-types are mostly programmer sensitive. More specifically, around 80% of the repeated additions as well as of the repeated modifications are programmer sensitive.

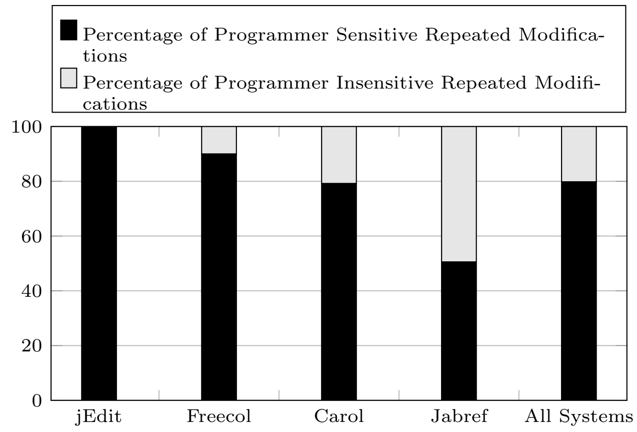

Figure 9: Percentage of Programmer Sensitive Repeated Modifications During Software Evolution.

Our findings from *RQ 2* can help us rank the change suggestions for a particular code snippet considering the programmer who is going to change that snippet. We can assume higher priorities for those change suggestions that are inferred from changes previously made by the working programmer. However, investigating such a ranking mechanism is not our main concern in this exploratory study. We plan to do this in the future.

5. RELATED WORK

Recommending changes to the programmers by analyzing the past evolution history of a software system is not new. Nguyen et al. [32] studied the repetitiveness of changes to the source code and found that small changes can be repeated at a high rate (70% to 100%) during software evolution. Repetition of changes decreases exponentially with the increase of change size. They implemented a change recommendation system exploiting the repetition tendencies of changes. They investigated the extent of change repetitions at different language constructs. However, in our study we investigate and compare the recommendation efficiencies for three types of changes: additions, deletions, and modifications. We also investigate the programmer sensitivity of repeated changes considering these three change types. Nguyen et al. [32] did not perform such investigations. Our investigations are important for priotizing and filtering change recommendations.

Ray et al. [37] developed a change recommendation system which stores and recommends changes that affect at least 50 tokens (7/8 lines). They did not perform a change type based analysis of recommendation efficiencies. In our study we investigate the recommendation efficiencies considering the three types of changes. We also investigate the programmer sensitivity of repeated changes. Ray et al. [37] did not perform this investigation.

In another study Nguyen et al. [31] investigated recurring bug-fixes as well as recurring changes. They implemented a system to identify code peers (i.e., code fragments that perform similar functionalities) and to automatically suggest changes to a code fragment where the changes were experienced by a peer code fragment. They use AST differencing algorithms for the purpose of change detection and recommendation. However, we use *diff* in our experiment. We also perform a change-type based analysis of the recommendation efficiencies and repetition tendencies of changes. Nguyen et al. [31] did not perform such investigations.

Toomin et al. [44] performed a study on simultaneous editing of multiple clone fragments in the working code-base of a software system. CloneTracker [9] also supports simultaneous editing of the clone fragments tracked by it. While these studies mainly deal with propagating a change that occurred in one clone fragment to a peer clone fragment, they do not deal with inferring future changes to a particular code fragment. Our study, on the other hand, deals with inferring change recommendations for any target code snippet whether it is a clone or not in the current revision.

There are also some studies [12,15,37] on the repetitiveness of code fragments in the software systems. A number of clone detection tools [5, 7, 40] also exist for detecting code reuse. In our study, we investigate the reuse of changes.

There are also a number of studies [1, 8, 17, 26–28, 47] that recommend peer code artifacts for co-changing (i.e., changing together) while changing a particular artifact on the basis of the past evolution history. However, our study is different. We do not recommend co-change artifacts. We deal with suggesting future changes to any code snippet.

A number of studies [2, 4, 14, 16, 29, 30, 34, 36, 41, 46] have also been done on code completion, particularly on method call completion or method body completion. In code completion, the programmer first writes a portion of the code and then, the completion engine provides suggestions to complete the rest of the code. Our study is different in the sense that we do not deal with the completion of the incomplete code. We deal with providing change recommendations for a target code snippet just after the programmer has clicked it. We perform a change-type based investigation on the recommendation efficiencies and repetition tendencies of changes.

In our empirical study, we address two important issues, repetition tendencies of the three different types of changes detected by UNIX *diff* and programmer sensitiveness of these three types of repeated changes, with interesting outcomes. Our investigation reveals the fact that repeated changes are mostly programmer sensitive. Consideration of this fact can help us develop a programmer sensitive change recommendation plug-in that can perform faster as well as provide a better ranking of the recommended changes.

6. THREATS TO VALIDITY

In our experiment we only consider the changes that occurred to the source code lines of the code fragments. We disregard the changes that occurred to the comments. We believe that comments are essential parts of a code-base. Sometimes commenting takes a longer time than writing code. However, in this research our primary aim was to deal with providing change suggestions to the code fragments, not to the comments. Thus, we disregard comments in our experiment. In future we plan to investigate on suggesting changes to the comments too.

We use the NiCad clone detector [6] in our experiment. While clone detectors suffer from the *confounding configuration choice problem* [45], NiCad has been found to perform fairly well [39,40]. Also, in a recent study [42] Svajlenko and Roy show that NiCad is a very good choice for detecting clones compared to other modern clone detectors.

The subject systems studied in this research are not enough to make a concrete decision about recommendation efficiencies and programmer sensitiveness of changes. However, our subject systems are of diverse variety in terms of application domains, system size (LOC), and the number of revisions. Thus, we believe that the outcome of our study cannot be attributed to chance. Our findings are useful for improving the existing change recommendation techniques to provide more precise recommendations.

7. CONCLUSION

Providing change recommendations to programmers on the basis of the past evolution history of a software system has been investigated by a number of existing studies. In our research we investigate two important issues regarding change recommendations: (1) the comparative effectiveness in recommending three types of changes - additions, deletions, and modifications, and (2) the programmer sensitivity of repeated changes considering those three change types. These two investigations are important for filtering and prioritizing change recommendations. The existing studies did not perform such investigations. From our in-depth investigation on thousands of commits of five diverse subject systems we can state that:

(1) Among the three change types (addition, deletion, and modification), modifications have the lowest tendency (1.3%) of being repeated. Possibly, a change recommendation tool should primarily focus on suggesting additions, and deletions disregarding the modifications.

(2) The same changes are generally implemented by the same programmers during software evolution. More specifically, around 71% of the repeated changes are programmer sensitive. Thus, when suggesting changes to a particular programmer we should prioritize those changes that were previously done by that programmer. A programmer sensitive change suggestion tool can help minimize the cognitive burden of the programmers in selecting the most suitable suggestion.

We believe that our findings are important for prioritizing as well as refining the change recommendations in order to get more precise recommendations. In future we would like to develop a change recommendation system considering our findings in this research.

8. REFERENCES

[1] A. Alali, B. Bartman, C. D. Newman, J. I. Maletic, "A Preliminary Investigation of Using Age and Distance Measures in the Detection of Evolutionary Couplings", Proc. *MSR*, 2013, pp. 169 – 172.

[2] M. Asaduzzaman, C. K. Roy, K. Schneider, D. Hou, "CSCC: Simple, Efficient, Context Sensitive Code Completion", Proc. *ICSME*, 2014, pp. 71 – 80.

[3] M. Asaduzzaman, C. K. Roy, K. Schneider, M. Di Penta, "LHDiff: A Language-Independent Hybrid

Approach for Tracking Source Code Lines", Proc. *ICSM*, 2013, pp. 230 – 239.

[4] M. Bruch, M. Monperrus, M. Mezini, "Learning from examples to improve code completion systems", Proc. *FSE*, 2009, pp. 213 – 222.

[5] CCFinderX: `http://www.ccfinder.net/ccfinderx.html`

[6] J. R. Cordy, C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC Tool Demo*, 2011, pp. 219 – 220.

[7] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, T. Xie, "XIAO: Tuning Code Clones at Hands of Engineers in Practice", Proc. *ACSAC*, 2012, pp. 369 – 378.

[8] E. Duala-Ekoko, M. P. Robillard,"Tracking Code Clones in Evolving Software", Proc. *ICSE*, 2007, pp. 158 – 167.

[9] E. Duala-Ekoko, M. P. Robillard, "CloneTracker: Tool Support for Code Clone Management", Proc. *ICSE*, 2008, pp.

[10] Effect Size: `http://en.wikipedia.org/wiki/Effect_size`

[11] Effect Size Calculation for Mann-Whitney-Wilcoxon Test: `http://www.let.rug.nl/~heeringa/statistics/stat03_2013/lect09.pdf`

[12] M. Gabel, Z. Su, "A study of the uniqueness of source code", Proc. *FSE*, 2010, pages 147 – 156.

[13] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 2012, 38(1):54 - 72.

[14] R. Hill, J. Rideout, "Automatic method completion", Proc. *ASE*, 2004, pp. 228 – 235.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. T. Devanbu, "On the naturalness of software", Proc. *ICSE*, 2012, pp. 837 – 847.

[16] D. Hou, D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion", Proc. *ICSM*, 2011, pp. 233 – 242.

[17] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard,"Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code", Proc. *WCRE*, 2010, pp. 119 – 128.

[18] M. Kim, D. Notkin, "Discovering and Representing Systematic Code Changes", Proc. *ICSE*, 2009, pp. 309 – 319.

[19] D. Kim, J. Nam, J. Song, S. Kim, "Automatic patch generation learned from human-written patches", Proc. *ICSE*, 2013, pp. 802 – 811.

[20] Mann-Whitney-Wilcoxon Test: `http://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test`

[21] Mann-Whitney-Wilcoxon Test Online: `http://www.socscistatistics.com/tests/mannwhitney/Default2.aspx`

[22] N. Meng, M. Kim, K. S. McKinley, "Sydit: Creating and applying a program transformation from an example", Proc. *ESEC/FSE*, 2011, pp. 440 – 443.

[23] N. Meng, M. Kim, K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples", Proc. *ICSE*, 2013, pp. 502 – 511.

[24] M. Mondal, C. K. Roy, K. A. Schneider, "Connectivity of Co-change Method Groups: A Case Study on Open-Source Systems", Proc. *CASCON*, 2012, pp. 205 – 219.

[25] M. Mondal, C. K. Roy, K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study", Proc. *ICPC*, 2013, pp. 103 – 112.

[26] M. Mondal, C. K. Roy, K. A. Schneider, "A Fine-Grained Analysis on the Evolutionary Coupling of Cloned Code", Proc. *ICSME*, 2014, pp. 51 – 60.

[27] M. Mondal, C. K. Roy, K. A. Schneider. "Prediction and Ranking of Co-change Candidates for Clones", Proc. *MSR* 2014, pp. 32 – 41 .

[28] M. Mondal, C. K. Roy, K. A. Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking", Proc. *SCAM*, 2014, pp. 11 – 20.

[29] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, T. N. Nguyen, "Graph-based pattern-oriented, context sensitive source code completion", Proc. *ICSE*, 2012, pp. 69 – 79.

[30] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, T. N. Nguyen, "A graph-based approach to API usage adaptation", *ACM Sigplan Notices*, 45(10): 302 – 321.

[31] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, T. N. Nguyen, "Recurring bug fixes in object-oriented programs", Proc. *ICSE*, 2010, pp. 315 – 324.

[32] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, H. Rajan, "A Study of Repetitiveness of Code Changes in Software Evolution", Proc. *ASE*, 2013, pp. 180 – 190.

[33] Nonparametric Tests: `http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Nonparametric/mobile_pages/BS704_Nonparametric4.html`

[34] T. Omori, H. Kuwabara, K. Maruyama, "A study on repetitiveness of code completion operations", Proc. *ICSM*, 2012, pp. 584 – 587.

[35] On-line SVN Repository: `http://sourceforge.net/`

[36] D. M. Pletcher, D. Hou, "BCC: Enhancing code completion for better API usability", Proc. *ICSM*, 2009, pp. 393 – 394.

[37] B. Ray, M. Nagappan, C. Bird, N. Nagappan, T. Zimmermann, "The Uniqueness of Changes: Characteristics and Applications", *Microsoft Research Technical Report*, 2014, pp. 1 – 10.

[38] C. K. Roy, J. R. Cordy, R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *Science of Computer Programming*, 2009, 74 (2009): 470 – 495.

[39] C. K. Roy, J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools", Proc. *Mutation*, 2009, pp. 157 – 166.

[40] C. K. Roy, J. R. Cordy, "Scenario-based Comparison of Clone Detection Techniques", Proc. *ICPC*, 2008, pp.153 – 162.

[41] R. Robbes, M. Lanza, "How Program History Can Improve Code Completion", Proc. *ASE*, 2008, pp. 317 – 326.

[42] J. Svajlenko, C. K. Roy, "Evaluating Modern Clone Detection Tools", Proc. *ICSME*, 2014, pp. 321 – 330.

[43] W. Takuya, H. Masuhara, "A Spontaneous Code Recommendation Tool Based on Associative Search", Proc. *SUITE*, 2011, pp. 17 – 20.

[44] M. Toomim, A. Begel, S. L. Graham, "Managing Duplicated Code with Linked Editing", Proc. *VL/HCC*, 2004, pp. 173 – 180.

[45] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation", Proc. *ESEC/FSE*, 2013, pp. 455 – 465.

[46] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, P. Ou, "Automatic parameter recommendation for practical API usage", Proc. *ICSE*, 2012, pp. 826 – 836.

[47] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, "Mining version histories to guide software changes", Proc. *ICSE*, 2004, pp. 563–572.