An Exploratory Study on Change Suggestions for Methods Using Clone Detection

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

A number of studies investigated providing change suggestions to programmers on the basis of the evolution history of a software system. While existing studies provide change suggestions considering code fragment level or even line level granularities, we investigate providing change suggestions at the method level. Providing a suggestion to change the entire method at one time is intuitively more time saving for developers compared to providing suggestions separately for different fragments of a method. In this research we empirically investigate whether we can infer change suggestions at the method level by analyzing the past evolution history of a software system through detection of method clones, and if so, then how we can rank the method level change suggestions.

According to our investigation on thousands of commits of seven diverse subject systems, we can provide change suggestions at the method level with up to 83% precision and 13.49% recall. Moreover, for up to 34% of the commits we can provide correct method level change suggestions. Compared to the existing fragment level change suggestion techniques, our method level change suggestion technique has promising precision and recall. We investigate the ranking of method level change suggestions and find that recency ranking (i.e., ranking on the basis of how recently the change suggestions appeared in the past) is a better choice than frequency ranking (ranking considering how frequently the suggestions appeared). We believe that while a method level change suggestion technique can never be a replacement for the existing fine grained change suggestion techniques, it can complement these existing ones.

1. INTRODUCTION

Repetition of changes is a common phenomenon during software maintenance and evolution. A change that occurred to a particular code fragment in the past can often be necessary for a similar code fragment in the future. A number of preexisting studies [25, 29, 11, 17, 24, 7] revealed this fact. Nguyen et al. [25] reports that repetitiveness of changes can be as high as 70-100% during system evolution. Researchers have developed change recommendation systems [25, 29] by exploiting the repetition tendencies of changes. The central idea behind these recommendation systems is simple and is explained as follows.

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 $CS_{current}$. The two existing change recommendation systems developed by Nguyen et al. [25] and Ray et al. [29] work according to this idea and can recommend changes with reasonable accuracy. Nguyen et al. and Ray et al. investigated different sets of subject systems and reported precisions of up to 30% and 59.91% respectively in providing change recommendations.

The existing change suggestion techniques [25, 29] provide code fragment level or even line level change suggestions. While such fine-grained change suggestions can perform well in helping programmers during development and maintenance, we were interested in improving the suggestion making capabilities of these techniques on the basis of the following facts.

(1) Changes might often need to be implemented in different parts/fragments of a method in a scattered way. The existing tecniques [25, 29] can provide suggestions for changing each fragment separately. However, getting suggestions for different fragments separately, identifying the most suitable suggestions, and applying them to the fragments one by one can often be very time consuming. In such a situation, it would be better if we could provide a complete method as a suggestion to the programmer such that the suggested method will be an updated version of the target method (i.e., the method that she wants to change) and will contain all the changes she wants to implement. In presence of such method level suggestions, a programmer can often avoid the difficulties of implementing multiple changes in multiple fragments of the target method in a time efficient manner.

(2) A particular code fragment is generally expected to be related to its context (i.e., the surrounding code). Two code fragments from two different methods might be similar. However, if the contexts of these two fragments are different,

CASCON'16, October 31-November 2, 2016, Markham, Toronto, Canada

DOI: 10.1145/1235

then the change that occurred to one fragment might not be suitable for the other one. If we can provide change suggestions at the method level, we can actually consider the contexts of the changes (i.e., the other parts of the method will work as the context). Thus, method level change suggestions are expected to be more precise than the fragment level suggestions. Our experimental results in Section 4.5 implies this.

Focusing on these facts, in our research we investigate the possibility of providing change suggestions considering method level granularity. When a programmer attempts to change a method (i.e., the target method), our goal is to provide her with suggestions where each suggestion will be an updated version of the target method containing the possible changes the programmer might want to implement.

When a target method is attempted to be changed we automatically analyze the method change history of the software system and identify whether any similar method (i.e., similar to the target method) was previously changed. The changed versions of such similar methods are considered as the change suggestions for the target method. For extracting the method change history of a software systen we detect and continuously update the genealogies of all the methods in the system as the system evolves through the commits. The genealogy of a particular method helps us understand how the method evolved through the commits by experiencing changes. We define method genealogy in Section 2.2.

For the purpose of our empirical study we implement our idea of providing method level change suggestions and apply the implementation on thousands of commits of seven diverse subject systems written in two different programming languages. We also investigate the ranking of method level change suggestions. In particular we answer the following two research questions.

- **RQ 1:** Can we infer change suggestions at the method level for changing a target method by analyzing the past evolution history of the software system?
- **RQ 2:** How can we rank method level change suggestions for a target method?

According to our experimental results and analysis we have the following findings:

(1) While the existing studies [25, 29] analyze and report the extent of fragment level change repetitions, we find that repetition of method level change is also a common phenomenon during system evolution. A target method might often need to be changed (including changes in multiple fragments) in the same way a similar method was changed in the past. According to our subject systems, up to 13.49% (recall) of the method changes occur in this way during evolution. Thus, method level change suggestions can also be inferred from the evolution history of a software system.

(2) Our precision in providing method level change suggestions can be as high as 83%.

(3) According to our investigated systems, we can provide correct method level change suggestions for up to 34% of the commit operations during the whole period of evolution. Thus, a method level change suggestion technique can often help programmers by providing correct change suggestions during software development and maintenance.

(4) While the existing techniques [25, 29] rank the fragment level change suggestions using frequency ranking (i.e., ranking on the basis of how frequently a suggested change occurred in the past), we find that recency ranking (i.e., ranking on the basis of how recently a suggested change occurred in the past) is a better choice than frequency ranking when ranking change suggestions at the method level.

With respect to the existing fine grained (i.e., fragment level) change recommendation studies [25, 29], our implemented change suggestion mechanism performs reasonably well (in terms of both precision and recall) in providing suggestions at the method level. Nguyen et al. [25] and Ray et al. [29] show that recall can be as low as 2% and 12.59% respectively. They studied fine grained change suggestions. They also report that recall decreases rapidly with the increase of fragment size. We investigate providing change suggestions at the method level which is a higher granularity compared to the fragment level granularity investigated in the existing studies. From this point of view we believe that our recall is reasonable. Our precision is promising, and is better compared to the existing studies. Thus, our technique can often provide accurate method level change suggestions. While a method level change suggestion mechanism can never be a replacement of the existing ones [25, 29], it can complement the existing techniques in providing better suggestions.

The rest of the paper is organized as follows. Section 2 describes our mechanism of providing method level change suggestions, Section 3 describes the experimental steps, Section 4 presents our investigation on the possibility of providing method level change suggestions and answers **RQ 1**, in Section 5 we investigate the ranking of the method level change suggestions and answer **RQ 2**, Section 6 describes related work, Section 7 discusses the possible threats to validity, and Section 8 concludes the paper.

2. PROVIDING CHANGE SUGGESTIONS AT THE METHOD LEVEL

In this section we describe the implementation of our idea of providing change suggestions at the method level. As we indicated in the introduction, method genealogy detection is an essential part in providing change suggestions at the method level. Ray et al.'s [29] approach does not involve the detection of method genealogies. Although Nguyen et al. [25] detect method genealogies, they provide change suggestions at the fragment level. Moreover, their genealogy detection technique, OAT [23], can only be used for Java systems [23]. The genealogy detection technique [18] that we use in our study is language independent. Table 1 shows that we also investigate a C# system in our study. In the following subsections we describe our mechanism of inferring method level change suggestions for a target method.

A Target Method. Let us assume that a programmer is going to make some changes to a method in the working code-base (i.e., the most recent code-base) of a software system. We call this method a *target method* in our paper.

2.1 Method Change Detection

In this research we consider the changes to the source code of a method disregarding the comments and indentations. Let us assume a particular method m_R in revision R. A commit operation C was applied to revision R, and revision $R+1$ was created because of this commit. The snap-shot of m_R in revision $R+1$ is m_{R+1}. We remove the comments and indentations from each of these two snap-shots (i.e., m_R, and m_{R+1}). Then we compare these two preprocessed snap-shots using UNIX *diff*. If the preprocessed snap-shots are different according to the *diff* output, then we consider that m_R was changed in the commit operation C.

2.2 Detection of Method Genealogies

Let us assume that a particular method was created in a particular revision of a subject system, and was alive in a number of consecutive revisions. Each of these revisions contains a snap-shot of the method. Detecting the genealogy of a particular method involves identifying the sequence of snap-shots of that method from those revisions where it was alive during system evolution. Our change suggestion mechanism involves the detection of method genealogies considering all the revisions of a candidate system. For detecting method genealogies, we follow the genealogy detection approach proposed by Lozano and Wermelinger [18].

From the genealogy of a particular method we can determine how it changed during the evolution of the software system by detecting the changes between every two consecutive snap-shots of it. Different methods might be created in different revisions. As a software system evolves through the commit operations, we extract as well as update the genealogies of all the methods in the system. We obtain the method change history of a software system by analyzing its method genealogies. We develop and continuously update a change suggestion database from the method change history of a candidate system as the system evolves.

2.3 Developing a Database for Providing Change Suggestions

For the purpose of suggesting changes we develop a database that contains the method change history of the past commits. We populate the database in such a way that we can easily mine it to infer change suggestions for changing a target method. At the very beginning of the development phase of a software system, this database remains empty. Each time a software system experiences a commit operation a new revision is created and we extract as well as update the genealogies of all the methods in the system. By analyzing these updated genealogies we determine which methods have been changed in this commit. We update the change suggestion database considering each occurrence of method change in this commit operation.

Let us assume that a particular method m has been changed in a commit operation C. From the genealogy of this method we retrieve the following two snap-shots of it and store these snap-shots in the database.

- **The prior snap-shot**. This is the snap-shot of the method m just before the commit operation C. We call this prior snap-shot m_{prior}.
- **The posterior snap-shot**. This is the snap-shot of m just after the commit operation C. We call this posterior snap-shot $m_{posterior}$.

We store these snap-shots in the database along with the commit number C. Thus, each entry in the database consists of three things: the two snap-shots, and the commit operation where the prior snap-shot was changed to the posterior snap-shot. In the following subsection we describe how we mine our change suggestion database for inferring method level change suggestions for changing a target method.

2.4 Inferring Change Suggestions for a Target Method from the Database

Let us assume that a programmer is now working on the code-base (i.e., the most recent code-base) of a software system. She identifies a particular method m_{target} where she needs to implement the changes. We call this method, m_{target}, the target method. Our goal is to provide her with a list of change suggestions for changing her target method m_{target} so that she can select a particular change suggestion and can apply the corresponding changes to m_{target}. We provide change suggestions at the method level by mining our change suggestion database. Each of our change suggestions can be regarded as an updated version of the target method containing all possible changes that the programmer might intend to implement in the target method. Our mechanism for inferring change suggestions for m_{target} is described below.

We mine the change suggestion database and determine whether the target method m_{target} is similar to any of the stored prior snap-shots. If m_{target} is similar to a prior snap-shot m_{prior}, then the corresponding posterior snap-shot $m_{posterior}$ is considered as a method level change suggestion for m_{target}. By mining the change suggestion database we determine all the prior snap-shots that are similar to m_{target}. The corresponding posterior snap-shots are considered as the method level change suggestions for m_{target}.

2.5 Detecting Similarity between two Methods

From our previous description we understand that detection of similarity between methods is an important part of providing change suggestions at the method level. We detect both exact and near-miss similarity between methods using the NiCad clone detector[5].

Exact similarity between methods. Let us consider two methods $m1$ and $m2$. If these two methods are the same disregarding the comments and indentations, we consider that the methods exhibit exact similarity.

Near-miss similarity between methods. Let us consider two methods $m1$ and $m2$. If these two methods have the same syntactic structure, however, the identifier names in one method are different compared to the other one, we consider that these two methods exhibit near-miss similarity.

We can easily understand that two methods that exhibit exact similarity also have the same syntactic structure. Thus, by considering the above two types of similarity we actually consider the syntactic similarity between methods. The existing change recommendation studies [25, 29] considered syntactic similarity between code fragments for providing fragment level change suggestions. We consider syntactic similarity between methods for providing method level change suggestions. We use NiCad [5] for detecting similarity in our research, because NiCad has been shown to detect the mentioned similarities (i.e., exact as well as near-miss similarities) with high precision and recall [31, 32, 35].

Table 1: Subject Systems

Systems	Lang.	Domains	LLR	SRev	ERev
jEdit	Java	Text Editor	191,804	3791	4000
GreenShot	C#	Screen Capturing Tool	37,628	1	999
Carol	Java	Game	25,091	1	1700
JHotDraw	Java	Graphics	143,339	1	500
Ant-Contrib	Java	Web Server	12,621	1	176
Java-ML	Java	Machine Learning Library	16,428	1	1200
DNSJava	Java	DNS Protocol	23,373	1	1635

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

3. EXPERIMENTAL STEPS

We perform our empirical study using our implemented change suggestion mechanism described in Section 2. We apply our implementation on the thousands of revisions of seven diverse subject systems listed in Table 1. We download these systems from an open-source SVN repository[1]. For each of the subject systems we perform the following experimental steps: (1) Download all the revisions of the subject system as mentioned in Table 1, (2) Extract methods from each of the revisions using Ctags[2], (3) Extract method genealogies considering all the revisions using the technique proposed by Lozano and Wermelinger [18], (4) Analyzing the method genealogies by comparing the consecutive snapshots using UNIX *diff* to identify the changes each method experienced in each commit operation during system evolution, (5) Build the change suggestion database considering the methods changed in each commit, and finally, (6) Investigate whether and to what extent we can provide method level change suggestions for changing a target method. We perform our investigation corresponding to Step-6 in Section 4. We investigate the ranking of method level change suggestions in Section 5.

4. INVESTIGATING METHOD LEVEL CHANGE SUGGESTIONS

In this section we answer the first research question by presenting and analyzing our experimental results.

RQ 1: *Can we infer change suggestions at the method level for changing a target method by analyzing the evolution history of the software system?*

Answering this question is the primary goal of our research. In the introduction we discussed that method level change suggestions can sometimes be more beneficial than fragment level suggestions. We answer RQ 1 by using our implemented change suggestion mechanism (Section 2).

4.1 Methodology

For answering RQ 1 we apply a variant of the *n-fold cross-validation* technique. We examine the commit operations of a subject system from the very first one. While examining a commit operation we first identify which methods changed in this commit. Then, we analyze whether we can infer these method-changes from the past evolution history of the system.

Let us assume that we are currently examining the commit operation C of a subject system. The commit C was applied on revision R and a method m_R was changed in this commit. We can easily determine which changes occurred to this method by retrieving the snap-shot m_{R+1} of the method in the next revision $R+1$. However, our goal is to determine whether and to what extent we can infer the snap-shot m_{R+1} by analyzing the method change history of the previous commits 1 to $C-1$ stored in the change suggestion database. Here, the method m_R is our target method.

We apply our change suggestion mechanism to mine the method change occurrences in the change suggestion database. We investigate all the method changes that occurred in the previous commits 1 to $C-1$. Let us consider that such a method change occurred in the commit operation $C-n$ (n >= 1). This commit was applied on revision $R-n$. The method m_{R-n} (i.e., the prior snap-shot) in this revision was changed in this commit. The snap-shot of the method in the next revision $R-n+1$ is m_{R-n+1} (i.e., the posterior snap-shot). We analyze whether this method change can be used for inferring a change suggestion for our target method m_R. We analyze in the following way.

- If we observe that the target method m_R is similar (similarity is defined in Section 2.5) to m_{R-n}, then we select m_{R-n+1} as the *change suggestion* for our target method m_R.
- If we observe that the target method m_R is similar to m_{R-n} and also, m_{R+1} is similar to m_{R-n+1}, then we consider m_{R-n+1} as the *correct change suggestion* for the target method m_R.
- If we see that the target method m_R is not similar to m_{R-n}, then we do not select m_{R-n+1} as the change suggestion for m_R.

In this way while examining a particular commit C, we analyze how many of the method changes that occurred in this commit can be correctly suggested by analyzing the method changes in the previous commits 1 to $C-1$.

4.2 Experimental Results

We analyze each occurrence of method change in each of the commits of a subject system using our change suggestion mechanism. For the purpose of explaining our experimental results we define a case in the following way.

A Case. A case is an occurrence of a method change. It consists of the following four things:

(1) A particular method m,

(2) A commit operation C where m was changed,

(3) The changed version $m_{changed}$ of the method m after the application of the commit operation C, and

(4) A list of method level change suggestions inferred by our change suggestion mechanism by analyzing the previous commits 1 to $C-1$ for changing the method m. The list of change suggestions might be empty. Let us consider a case with a non-empty list of method level suggestions. If a suggested method $m_{suggested}$ (i.e., a method level suggestion) from the list is similar to $m_{changed}$, then we understand that our mechanism can provide the correct change suggestion for changing m in this case.

We identify all the cases from the entire evolution history of a subject system and determine the following three sets.

- **AC (All Cases):** *This set includes each of the cases that we identify by examining the evolution history of a subject system. In other words, this set includes all the*

[1]Open-Source SVN Repository: http://www.sourceforge.net
[2]Exhuberant Ctags: http://ctags.sourceforge.net/

Table 2: Statistics for Method Change Suggestions

	jEdit	Greenshot	Carol	JHotDraw	Ant Contrib	Java-ML	DNSJava
NAC	2616	1999	2864	8335	60	1778	3770
NCS	543	262	246	1820	6	328	513
NCCS	353	113	74	822	5	145	157

NAC = No. of Cases in the set AC. AC is defined in Section 4.2.
NCS = No. of cases where we can provide one or more change suggestions. This is the number of cases in the set CS.
NCCS = No. of cases where the suggestions that we provide include the correct change suggestion (i.e., no. of cases in the set CCS).

occurrences of method change. If n methods experience changes (additions, deletions, and/or modifications) in a particular commit, then we identify n cases from this commit.

- **CS (Cases with Suggestions):** Each case for which we can provide one or more change suggestion is included in this set. Let us consider a case where a method m was changed in a commit operation C. If our method change suggestion mechanism can infer one or more change suggestions for changing m by analyzing the previous commits 1 to $C-1$, then we include this case in the set **CS**.

- **CCS (Cases with Correct Suggestions):** Each case where the change suggestions provided by our mechanism include the *correct suggestion* is included in this set. **CCS** is a proper subset of **CS**.

The number of cases in each of the above sets from each of our subject system is shown in Table 2. On the basis of these sets we present our analysis in the following subsections.

4.3 Analyzing the percentage of repeated cases (i.e., the repetitiveness of method change)

We can easily understand that if the change suggestions inferred by our mechanism for a particular case include the correct suggestion (i.e., if this case was included in the set **CCS**), then this case is an example of a repeated case. We define a repeated case in the following way.

A Repeated Case. Let us consider a case $case_{current}$ where a method $m_{current}$ was changed to $m_{current,changed}$ because of the commit operation $C_{current}$. We also consider another case $case_{previous}$ from a previous commit $C_{previous}$ where a method $m_{previous}$ was changed to $m_{previous,changed}$. We say that $case_{current}$ is a repetition of $case_{previous}$ if $m_{current}$ is similar to $m_{previous}$, and also, $m_{current,changed}$ is similar to $m_{previous,changed}$.

An Example of a Repeated Case. Fig. 1 shows an example of a repeated case from our subject system JavaML. The changes experienced by a method named *paintComponent* from class *VisualTestXMeans.ClusterLabel* (file = VisualTestXMeans.java) in commit operation 52 were also experienced by another similar method with the same name from class *VisualTestSimpleKMeans.ClusterLabel* (file = VisualTestSimpleKMeans.java) in commit operation 61. We see that the snap-shots of the method in *VisualTestXMeans.ClusterLabel* just before and after the commit operation 52 are similar to the corresponding snap-shots of the method in class *VisualTestSimpleKMeans.ClusterLabel* just before and after commit 61.

Providing the correct change suggestion for a particular case is possible only if this case is a repeated case. The percentage of repeated cases during the whole period of evolution of a software system was calculated using Eq. 1.

$$\%\ of\ Repeated\ Cases = Recall = \frac{|CCS| \times 100}{|AC|} \quad (1)$$

The percentage of repeated cases for each of our subject sytems is shown in Table 3. This percentage is the highest (13.49%) for our candidate system jEdit.

We see that the percentages regarding some of the subject systems (such as Carol, GreenShot) are low. The primary reason behind such low percentages is the granularity of change suggestions that we infer. We provide a whole method as a suggestion. When a change that occurred to a particular fragment of a previous method can be applicable to a similar fragment of the target method, the whole target method might not replaceable by the changed version of the previous method. However, although the percentages of repeated cases are low for some subject systems, we see that this percentage can sometimes be considerable (For example, our system jEdit). Moreover, these percentages are reasonable when compared with the existing change recommendation studies [25, 29].

Nguyen et al. [25] reports that the percentage of repeated changes can be as high as 70-100% for small changes. This percentage drops exponentially with the increase of change size. For large changes, the repetitiveness is below 2%. They measured the extent of fragment level change repetitions in their study. In other words, they determined the extent to which a change in a code fragment is repeated. However, in our study we measure the repetitiveness of *cases*, that is the repetitiveness of the whole method change. In a repeated *case*, there can be repetitions of changes to multiple code fragments remaining inside a method. Thus, the percentage of repeated cases will reasonably be smaller than the percentage of repeated fragment level changes. However, our experimental results show that the percentage of repeated cases can often be considerable. Thus, method level change suggestions has the potential to help programmers during software evolution.

Ray et al. [29] investigate the repetitiveness of changes in three subject systems. The percentages of repeated changes in their systems were 17.44%, 25.18%, and 12.59% respectively. Their granularity of investigation was in the fragment level. They only considered fragments of at least 50 tokens. The changes that occurred to any smaller fragments were disregarded in their study. On the other hand we investigate the repetitiveness of cases (i.e., repetition of the occurrence of a whole method change), and also consider all the changes that occurred during evolution. With such differences in considerations we believe that the percentages of repeated cases in our subject systems are reasonably well.

We finally believe that the oftenness of the repetition of *cases* is considerable during software evolution. Thus, change suggestions at the method level can be helpful to the programmers during development and maintenance.

We can also easily understand that the percentage of repeated cases (calculated using Eq. 1) is the recall in providing change suggestions in the method level. We call it recall

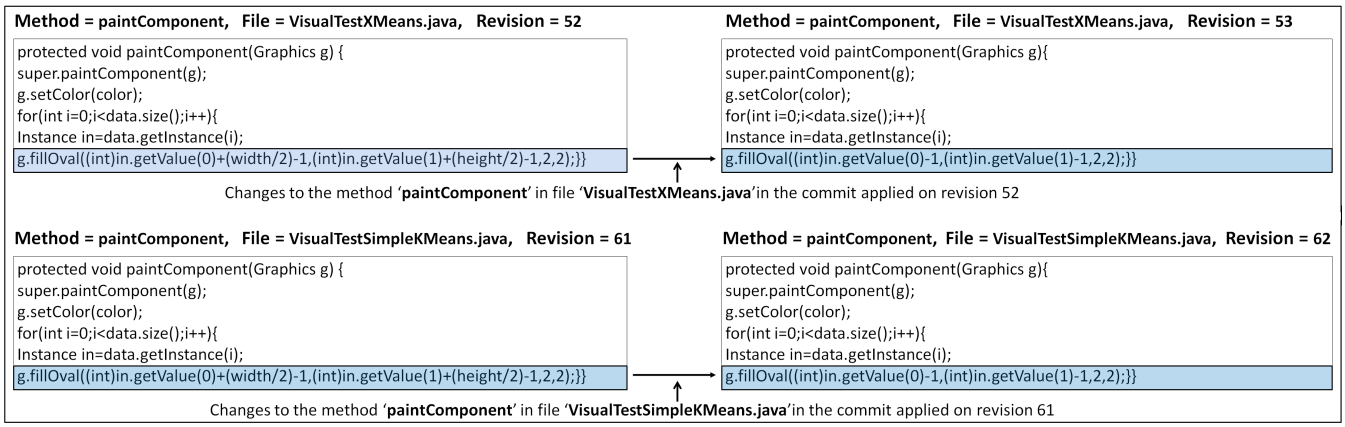

Figure 1: The figure shows an example of a repeated case from our subject system JavaML. We see that the changes that occurred to a method in the commit applied on revision 52 were also experienced by a similar method in the commit on revision 61.

Table 3: The percentage of repeated cases

	jEdit	Greenshot	Carol	JHotDraw	Ant Contrib	Java-ML	DNSJava
PRC	13.49%	5.65%	2.58%	9.86%	8.33%	8.16%	4.16%

PRC = Percentage of repeated cases.

because it is the percentage of those cases (method changes) for which we can provide accurate method level change suggestions with respect to all cases. Thus we believe that our recall in providing method level change suggestions is reasonable compared to the existing studies.

4.4 Manual analysis of the repeated cases (i.e., the cases in the set CCS)

We also manually analyze the repeated cases to determine whether the changes that were previously experienced by a particular method in a commit operation $c_{previous}$ can again occur to a similar method in a later commit operation c_{later}. In the previous subsection we explained that the cases in the set CCS are the repeated cases. We manually analyze 20 such cases from each of the subject systems except Ant-Contrib. For this system, the total number of cases in CCS is 5. From our analysis on 125 cases in total we can state that *in each of the cases the changes were repeated*. We realize that *the changes that previously occurred to a particular method can again occur to a similar method in future*. Fig. 1 shows an example of a repeated case from the subject system JavaML.

4.5 Analyzing the precision in providing method level change suggestions

Precision is the percentage of cases where the change suggestions provided by our implemented mechanism include the correct change suggestion with respect to all those cases where we can provide one or more suggestions. We calculate precision in the following way.

$$Precision = (|CCS| \times 100) \ / \ |CS| \quad (2)$$

Fig. 2 shows how precisely we can provide method level change suggestions for each of our subject systems. We see that the precision is considerable for each of our candidate systems. Considering all the subject systems we also calculate the overall precision using the following equation.

$$Overall\ Precision = \frac{100 \times \sum_{s\ \epsilon\ Systems} |CCS_s|}{\sum_{s\ \epsilon\ Systems} |CS_s|} \quad (3)$$

Here, s is a particular subject system, and CCS_s and CS_s are respectively the sets CCS and CS from this system.

The overall value of precision is around 44.88% in our study. In case of Nyguen et al.'s study [25], this overall value was around 30%. Ray et al. [29] reported a precision of up to 59.91% in their study. However, in our study we see that this precision can be as high as 83% (for Ant-Contrib). For our largest system jEdit, the precision is 65%. Although our change suggestion mechanism is not directly comparable with the existing ones [25, 29] because of the difference between the granularities of change suggestion, we believe that our mechanism performs reasonably well in providing change suggestions at the method level. Also, as the suggestions provided by our mechanism are full methods, a programmer can often readily use (i.e., using without any change) the correct suggestion (i.e., the correctly suggested method that already contains the expected changes) by replacing the target method she attempted to change.

4.6 Analyzing the percentage of cases where we can provide one or more change suggestions at the method level

We calculate this percentage using the following equation.

$$PCS = (|CS| \times 100) \ / \ |AC| \quad (4)$$

Here, PCS is the percentage of cases where we can provide one or more change suggestions. The PCS for each of the subject systems has been shown in Fig. 3.

Fig. 3 implies that the percentage of cases where we can provide method level change suggestions is considerable for most of the subject systems. The percentage regarding JHotDraw (i.e., 21.84) is the highest. We also calculated the overall percentage considering all subject systems using an equation similar to Eq. 3. This overall value is around

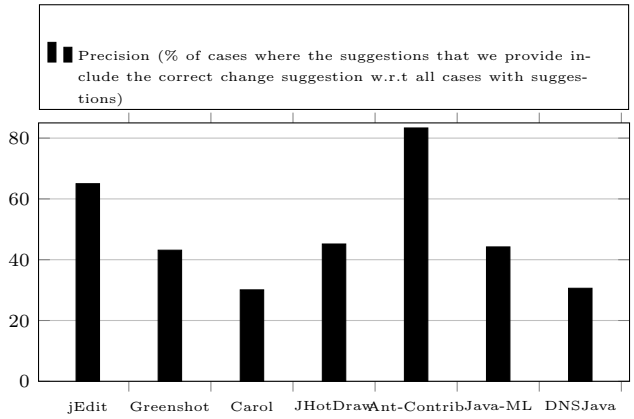

Figure 2: Precision in providing method level change suggestions

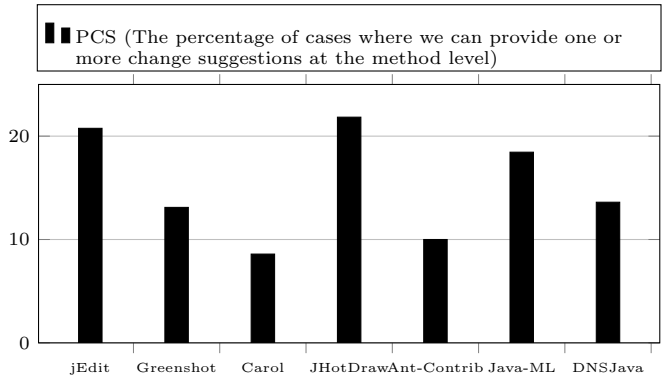

Figure 3: Percentage of cases where we can provide one or more change suggestions at the method level

17.36%. We believe that our implemented mechanism has the potential to help programmers by providing method level change suggestions during development and maintenance, and thus, can complement the existing fine-grained change suggestion techniques [25, 29].

4.7 Analyzing the percentage of commit operations with correct change suggestions

This is the percentage of commits where we can provide the correct change suggestions at the method level with respect to all commits. This percentage can help us understand how often our change suggestion mechanism can help programmers by providing them the correct suggestions at the method level. A number of methods might be changed in a particular commit operation. If we can provide correct change suggestion for any of these methods, then we consider this commit for our calculation. We calculate this in the following way.

$$POCCS = (|CCCS| \times 100) \ / \ |CMC| \quad (5)$$

Here, *POCCS* is the percentage of commits where we can provide correct change suggestions at the method level. *CMC* is the set of those commits where one or more methods were changed. The set *CCCS* includes each of those commits where we can provide the correct change suggestion(s) at the method level. We plot the percentage *POCCS* for each of the subject systems in the graph of Fig. 4. For our subject system JHotDraw, this percentage is the highest (34%). We also calculate the overall value of *POCCS* considering all the subject systems using an equation similar to Eq. 3. This overall value is around 11%. From our experimental results we can say that our change suggestion mechanism can often provide correct suggestions (i.e., at the method level) to the programmers during system evolution.

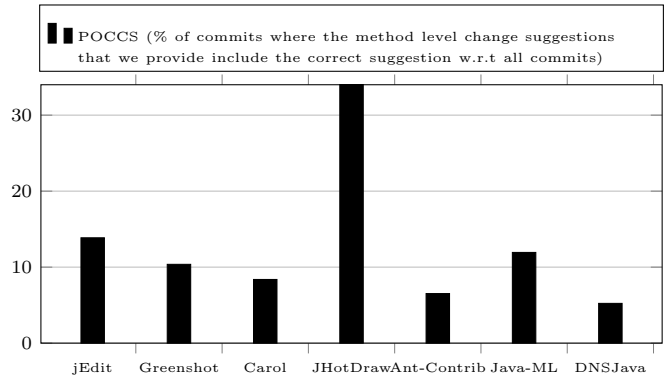

Figure 4: The percentage of commits where the change suggestions that we provide include the correct suggestion at the method level.

4.8 Answer to RQ 1

According to our analysis involving manual investigation in the previous subsections we can state that:

> ***(1)*** *Repetition of method change is a common phenomenon during system evolution. A target method might often need to be changed in the same way a similar method was changed previously. According to our investigated systems, up to 13.49% of the method changes occur in this way during system evolution. This percentage is the recall in providing method level change suggestions. The overall recall considering all subject systems is around 7.8%. With respect to the existing fine grained change suggestion techniques [25, 29], our method level change suggestion technique shows a reasonable recall value for each of our subject systems.*
>
> ***(2)*** *Our implemented change suggestion technique can provide change suggestions at the method level for up to 21.8% of the cases. Our precision in providing method level change suggestions can be as high as 83%. The overall precision is around 44.89% considering all candidate systems.*
>
> ***(3)*** *The overall percentage of commits where we can provide correct change suggestions is also considerable (11%). This percentage can be as high as 34% according to our investigated systems. Thus, change suggestions at the method level can often help programmers in making changes during software evolution.*

A method level change suggestion inferred by our implemented mechanism can contain multiple changes at multiple fragments of it. By using UNIX *diff* or other existing program differentiation tools [16, 3] we can easily identify the differences between a target method and a suggested

Table 4: Average number of method level change suggestions per target method

	jEdit	Greenshot	Carol	JHotDraw	Ant-Contrib	Java-ML	DNSJava
ANS	1.59	2.08	1.86	7.02	1.5	1.99	2.97

ANS = Average number of change suggestions at the method level.

Table 5: Frequency and recency ranking examples

	S1	S2	S3	S4	S5	S6	S7
Frequency	2	2	1	3	4	4	4
Recency	101	190	103	190	98	101	74
Frequency based ordering	S5	S6	S7	S4	S1	S2	S3
Recency based ordering	S2	S4	S3	S1	S6	S5	S7

S1 to S7 are the distinct change suggestions.
Recency is the most recent commit operation where a change suggestion occurred.

method. Here, we should note that there can be situations where some of the changes in a method level change suggestion might not be suitable for the target method. However, even in such a situation the suggestion can be helpful to the programmer. She can consider only those changes which are suitable for her target method disregarding the remaining ones. She can see her intended changes implemented in a similar context as of her target method.

5. RANKING OF CHANGE SUGGESTIONS

In this section we answer the second research question (RQ 2) by presenting and analyzing our experimental results.

RQ 2: *How can we rank method level change suggestions for a target method?*

From our answer to the previous research question we see that we can provide change suggestions at the method level for changing a target method by analyzing the software evolution history. In RQ 2 we investigate how we can rank the method level change suggestions provided by our technique. We perform our investigation in the following way.

5.1 Investigating the average number of change suggestions per target method

Table 4 shows the average number of change suggestions per target method for each of our subject systems. We see that for our subject system JHotDraw this average number is the highest (7.02). Although the average number of suggestions for some systems are small, we found that the number of change suggestions for some target methods could be high. For jEdit, GreenShot, Carol, Java-ML, and DNSJava the highest numbers of suggestions for a target method can be 8, 10, 19, 17, and 12 respectively. From such a scenario we believe that efficient ranking of the method level change suggestions is important. In the following subsections we discuss the ranking of change suggestions.

5.2 Ranking of change suggestions

We discuss and compare the following two ranking mechanisms for ranking the method level change suggestions for a target method.

- **Frequency Ranking:** Ranking change suggestions on the basis of their frequencies of occurrences.
- **Recency Ranking:** Ranking change suggestions on the basis of their recency of occurrence.

The existing studies [25, 29] only used frequency ranking for ranking change suggestions. However, their suggestions were fine grained (i.e., at the fragment level). We provide suggestions at the method level. In our study we investigate whether frequency ranking or recency ranking can be a better choice for ranking method level suggestions.

Frequency Ranking. Let us assume that our change suggestion technique obtains a number of method level change suggestions for a target method by analyzing the past evolution history. However, more than one change suggestions retrieved from the same or different commits might be similar (similarity is defined in Section 2.5) to one another. We first identify the distinct suggestions. For each of these distinct suggestions we determine how many times it occurred in the past evolution history. We call this number the frequency of occurrence of the suggestion. We sort the distinct change suggestions in decreasing order of their frequencies. In *frequency ranking* we assume higher priorities for those suggestions that occurred more frequently in the past compared to the others.

Recency Ranking. In this ranking mechanism we first identify the distinct change suggestions, and then, for each distinct suggestion we determine the last commit operation where it occurred. We call this last commit the recency of the suggestion. We sort the suggestions in decreasing order of their recency. In *recency ranking* change suggestions with higher recency values (i.e., the change suggestions that occurred more recently) are given higher priorities compared to the others. In a previous study [20] we used recency ranking for ranking the co-change candidates for clones.

Examples of frequency and recency ranking. Table 5 shows the ranking of seven distinct change suggestions, S1 to S7. We can see the frequency as well as the recency of each suggestion. In the frequency based ordering we see that the suggestions: S5, S6, and S7 with the highest frequency (i.e., 4) come first (i.e., at the left). In case of recency based ordering, the most recent suggestions (i.e., S2, and S4) have been given the highest priorities.

5.3 Comparing frequency ranking and recency ranking.

We compare frequency and recency ranking mechanisms for ranking method level change suggestions provided by our change suggestion technique. For the purpose of our description we define an *eligible case* in the following way.

An Eligible Case. Let us consider a case (we provide the definition of a case in Section 4) where the method level suggestions provided by our change suggestion technique include the correct suggestion. We call this case an *eligible case*. For the purpose of comparing frequency and recency ranking mechanisms we investigate all the eligible cases of a subject system.

Comparing ranking mechanisms using an eligible case. Let us consider an eligible case. We order the list of non-empty change suggestions corresponding to this case using each of the two ranking mechanisms. We can easily understand that the ranking mechanism that assigns a better rank to the correct change suggestion should be considered the superior one for this eligible case. We explain this using a very simple example.

Fig. 5 shows a list of six change suggestions: S1 to S6.

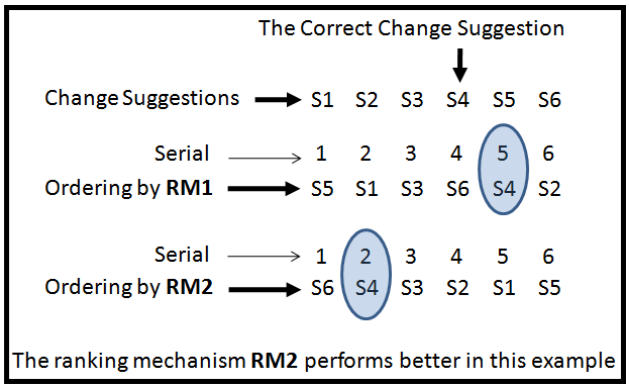

Figure 5: Comparing the ranking mechanisms, RM1 and RM2

We identify the correct suggestion. We also show two possible orderings of these suggestions using two ranking mechanisms: RM1 and RM2. We see that in the ordering performed by RM1, the correct suggestion has a serial number of 5. Also, in the ordering performed by RM2, the correct suggestion has a serial number of 2. Thus, in this example RM2 is superior to RM1, because RM2 assigns a higher priority (smaller serial) to the correct change suggestion compared to RM1.

Comparing frequency and recency ranking. By examining the evolution history of a subject system we identify all the eligible cases and determine the following measures by analyzing these cases.

- The number of cases where frequency ranking assigns better ranks to the correct change suggestions compared to recency ranking.
- The number of cases where recency ranking assigns better ranks to the correct change suggestions compared to frequency ranking.
- The number of cases where both ranking mechanisms assign the same rank to the correct suggestions.

We show the above measures in Table 6. We see that for most of the eligible cases of each of the subject systems, frequency ranking and recency ranking provide the same rank to the correct change suggestion. Fig. 6 shows a visual comparison between the percentages of cases where frequency ranking or recency ranking performs better. We see that for three subject systems: jEdit, Carol, and Java-ML recency ranking performs better than frequency ranking. For each of the eligible cases of Ant-Contrib, both ranking mechanisms assigned the same rank to the correct suggestion. For GreenShot the percentages regarding the two ranking mechanisms are the same. For two systems, JHotDraw and DNSJava, frequency ranking performs better than recency ranking. We also determined the overall values of the two percentages for each of the subject systems. We find that recency ranking performs better than frequency ranking for overall 3.22% higher number of the cases. From such a scenario we can decide that in general recency ranking is better than frequency ranking for ranking method level change suggestions.

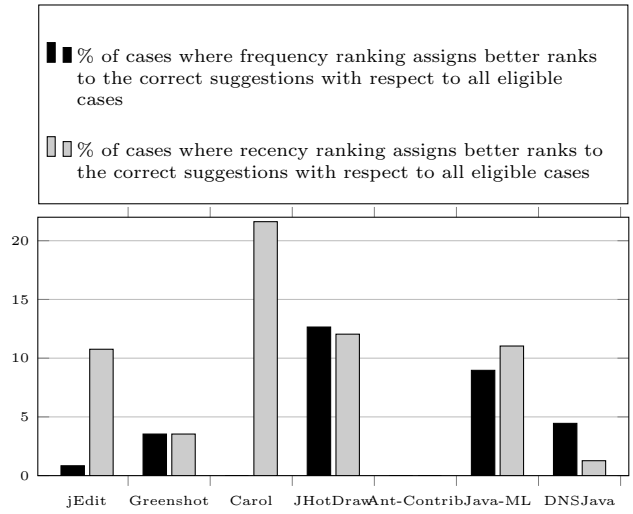

Figure 6: Comparing frequency ranking and recency ranking.

5.4 Answer to RQ 2

From our analysis and discussion in the previous subsections we come to the following conclusion.

> *Recency ranking is possibly a better choice compared to frequency ranking when ranking change suggestions at the method level.*

6. RELATED WORK

Change recommendation by exploiting the repetition tendencies of changes during system evolution has been investigated by a number of studies.

Nguyen et al.[25] investigated fragment level change repetitions. They developed a change recommendation technique that can provide fragment level change suggestions to a programmer when she attempts to change a code fragment. They ranked the fragment level suggestions using frequency ranking. In our study we investigate method level change repetitions and develop a change recommendation system that can provide change suggestions at the method level. We also show that recency ranking is better than frequency ranking in case of ranking method level change suggestions.

Ray et al. [29] also developed a change recommendation system considering fragment level granularity. They ranked the fragment level change suggestions using frequency ranking as was done by Nguyen et al. [25]. In our study we develop a method level change recommendation system. From our investigation on the ranking of method level suggestions we see that recency ranking performs better than frequency ranking.

In another study Nguyen et al. [24] 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. In our study we investigate method level

Table 6: Statistics regarding ranking of change suggestions using frequency and recency ranking

	jEdit	Greenshot	Carol	JHotDraw	Ant-Contrib	Java-ML	DNSJava
F > R	3	4	0	104	0	13	7
R > F	38	4	16	99	0	16	2
R = F	312	105	58	619	5	116	148

F > R = No. of cases where frequency ranking is better than recency ranking.

R > F = No. of cases where recency ranking is better than frequency ranking.

'R = F' is the no. of cases where both ranking provides the same rank.

change recommendations. We use *diff* in our experiment for change detection. Nguyen et al. did not use any ranking mechanism for ranking the change suggestions. We investigate frequency ranking and recency ranking for ranking our method level change suggestions, and find recency ranking to be the better one.

Toomim et al. [37] 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 cannot infer which changes might occur in a particular code fragment in future. Moreover, they do not deal with any ranking mechanism. Our study, on the other hand, deals with inferring method level change suggestions for any target method whether it is a clone or not in the current revision. We also investigate the ranking of change suggestions.

There are also some studies [13, 29, 10] on the repetitiveness of code fragments in the software systems. A number of clone detection tools [28, 6, 5, 33] also exist for detecting code reuse. In our study, we investigate the reuse of changes.

There are also a number of studies [8, 15, 1, 40, 19, 20, 21] 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 recommend possible future changes to a target method.

A number of studies [2, 12, 39, 4, 14, 34, 27, 26, 22, 23] 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. Our goal is to provide change suggestions for a target method just after the programmer has clicked it. Our change suggestions are the changed versions of the previously changed similar methods (similar to the target).

Our experimental results imply that we can infer method level change suggestions for a target method with comparable accuracy with respect to the accuracies reported in the existing studies [25, 29]. We believe that our method level change suggestion mechanism can complement the existing fine grained change suggestion techniques in providing better change suggestions to programmers.

7. THREATS TO VALIDITY

We use the NiCad clone detector [5] in our experiment. While clone detectors suffer from the *confounding configuration choice problem* [38], NiCad has been found to perform fairly well [32, 33]. Also, in a recent study [35] Svajlenko and Roy show that NiCad is a very good choice for detecting clones compared to other modern clone detectors.

We provide change suggestions at the method level. For long methods it might be difficult for the programmers to identify the differences between a target method and a suggested method. However, different program differentiation tools [16, 3] are currently available. These tools or even UNIX *diff* can help us identify the differences easily.

The subject systems studied in this research are not enough to make a concrete decision about method level change suggestions and their ranking. 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, and our findings are significant.

8. CONCLUSION

In this paper we present our study on method change repetitions during system evolution. By exploiting the repetition tendencies of method changes we develop a change suggestion mechanism that can provide change suggestions at the method level. According to our experimental results on thousands of commits of seven diverse subject systems, we can provide method level change suggestions for up to 21.8% of the cases (i.e., method changes). Our precision and recall in providing method level change suggestions can be as high as 83% and 13.49% respectively. The overall values of these two measures (i.e., precision and recall) are around 44.89% and 7.8%. With respect to the existing studies our accuracy in providing change suggestions is reasonable. Method level change suggestions can often be useful to programmers during software development and maintenance, because for up to 34% of the commits our mechanism provides correct method level change suggestions.

We also investigate the ranking of method level change suggestions and find that recency ranking performs better than frequency ranking. Finally, we believe that while method level change suggestions can never be a replacement of the fine grained (i.e., fragment level) change suggestions, our change suggestion mechanism can complement existing techniques in providing better suggestions.

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

[6] 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.

[7] B. Dagenais, M. P. Robillard, "Recommending adaptive changes for framework evolution", Proc. *ICSE*, 2008, pp. 481 – 490.

[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. 843 – 846.

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] A. Lozano, M. Wermilenger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227 - 236.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] D. Rattan, R. Bhatia, M. Singh, "Software Clone Detection: A Systematic Review", *Information and Software Technology*, 2013, 55(7): 1165 – 1199.

[29] 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.

[30] C. K. Roy, J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. *ICPC*, 2008, pp. 172 – 181.

[31] 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.

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

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

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

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

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

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

[38] 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.

[39] 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.

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