


Article

A Change Recommendation Approach Using Change Patterns of a Corresponding Test File

Jungil Kim ¹  and Eunjoo Lee ^{2,*}

¹ Department of Software Technology Laboratory, Kyungpook National University, Daegu 41566, Korea; jikim424@gmail.com

² School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Korea

* Correspondence: ejlee@knu.ac.kr; Tel.: +82-53-950-7548

Received: 31 August 2018; Accepted: 19 October 2018; Published: 23 October 2018



Abstract: Change recommendation improves the development speed and quality of software projects. Through change recommendation, software project developers can find the relevant source files that they must change for their modification tasks. In an existing change-recommendation approach based on the change history of source files, the reliability of the recommended change patterns for a source file is determined according to the change history of the source file. If a source file has insufficient change history to identify its change patterns or has frequently been changed with unrelated source files, the existing change-recommendation approach cannot identify meaningful change patterns for the source file. In this paper, we propose a novel change-recommendation approach to resolve the limitation of the existing change-recommendation method. The basic idea of the proposed approach is to consider the change history of a test file corresponding to a given source file. First, the proposed approach identifies the test file corresponding to a given source file by using a source–test traceability linking method based on the popular naming convention rule. Then, the change patterns of the source and test files are identified according to their change histories. Finally, a set of change recommendations is constructed using the identified change patterns. In an experiment involving six open-source projects, the accuracy of the proposed approach is evaluated. The results show that the accuracy of the proposed approach can be significantly improved from 21% to 62% compared with the existing approach.

Keywords: change coupling; change recommendation; data mining; open source project; github

1. Introduction

Software systems constantly evolve to improve their quality and extend their lifetime [1–3]. As a software system evolves, its source files are inevitably changed [4–8]. A new source file is added, and the code of existing source files is modified. In modification tasks of software project developers, a change of a source file may affect other source files according to dependent relationship among the source files. If such change impacts cannot be immediately taken into account, software project developers may face unexpected errors in the near future [9,10]. Hence, software project developers should catch all the change impacts related to their modification tasks to avoid unexpected errors and reduce maintenance costs of a software system. However, it is typically difficult to manually identify all change impacts related to particular changes of source files in development of a large software system.

Change recommendation can reduce the efforts for identifying the change impacts [11–15]. Through change recommendation, software project developers can immediately identify relevant source files that they must change. The change-recommendation approach is based on a static analysis or a change-history analysis. The basic idea of change-history analysis for change recommendation

originates from the concept of association rule discovery [16–18]. Association rule discovery is a data mining method and an unsupervised machine learning algorithm. Its objective is to extract association patterns between items in a large dataset.

In previous studies [11,12], a change-recommendation approach based on association rule discovery was proposed. Given a source file as a query for change recommendation, the change-recommendation approach analyzes the change history of the source file to derive the change patterns of the source file. In the change-recommendation approach, a change pattern indicates a change association between two source files, and the reliability of a change pattern is determined by change coupling between source files. The change coupling is typically computed by the co-change frequency of source files. For example, two source files have high change coupling if they have been frequently changed together. In contrast, two source files have low change coupling if they have been rarely changed together. According to this concept, the change recommendation approach requires sufficient and also clear change history for source files as much as possible to identify meaningful change patterns. This constraint is a major cause of the degradation of the applicability of the change-recommendation approach [19–21]. For example, if a source file has insufficient change history that is very short to identify its change patterns, meaningless change patterns of the source file may be identified. In addition, if a source file has been accidentally changed with functionally unrelated source files, the reliability of the change pattern extracted from the change history cannot be guaranteed in the change recommendation [22,23].

In this study, the co-evolution relationship between source and test files is taken into account to resolve the limitation of the existing change-recommendation approach. Generally, in software system development, source files and their corresponding test files evolve together [24–27]. When a new source file is added or an existing source file is modified, the test files related to the added or modified source files are modified to validate the added and modified code [28]. A test file is obligated to test several related source files. Hence, we believe that some of the closely related change patterns of a source files can also be identified in the change history of the corresponding test file. The existing change-recommendation approach only considers the change histories related to given source files, not the change histories of the test file corresponding to the given source files.

Based on the aforementioned idea, a novel change-recommendation approach is proposed. The proposed change-recommendation approach considers not only the change history of a given source file, but also the change history of a test file corresponding to the given source file to make a change recommendation. Generally, in software development, source files are paired with their corresponding test files according to a specific naming convention rule [29]. The naming convention rule can be often used to trace pairs of source and test files that are explicitly related [30]. Given a source file as a query for change recommendation, the proposed approach first identifies a test file corresponding to the given source file by using a source–test traceability linking method based on the naming convention rule. Then, the change histories related to the source and the test files are extracted from a source-code repository. Finally, the change patterns of the source and the test files are identified from the extracted change histories, and then a set of change recommendations is constructed using the change patterns. In our experiment involving six open-source projects, we evaluated the recommendation accuracy of the proposed change-recommendation approach. The experimental result shows that the accuracy of the proposed change-recommendation approach can significantly improve the accuracy of the existing change-recommendation approach in the open-source projects. On average, the proposed change-recommendation approach improved the accuracy from 21% to 62% compared with the existing change-recommendation approach. Therefore, we believe that the proposed change-recommendation approach is useful for real-world software project developments.

The remainder of the paper is organized as follows. Related works are introduced in Section 2. The proposed change-recommendation approach is described in Section 3. The experimental results are reported and discussed in Section 4. The study is concluded in Section 5.

2. Related Work

2.1. Association Rule Discovery and Change Recommendation

Association rule discovery is a data-mining method for identifying meaningful association patterns from a large-scale data set [16,17,31]. An association pattern indicates a specific association between items [32]. A representative example of an association pattern is a purchasing pattern on an online shopping site, such as Amazon.com. For example, a common purchasing pattern between televisions and Blu-ray players can be inferred from the fact that televisions and Blu-ray players are frequently purchased together by customers on Amazon.com. The common purchasing pattern indicates that in general, customers who purchase a television also want to purchase a Blu-ray player. Amazon.com can use this pattern to recommend a Blu-ray player to customers who add a television product to their shopping cart.

Formally, an association pattern in association rule discovery is defined as follows:

Given a set of items $I = \{I_1, I_2, \dots, I_n\}$ and a set of database transactions $T = \{t_1, t_2, \dots, t_m\}$, where n and m are the total numbers of items and transactions, respectively, an association pattern between two item sets is formed as $\{A \rightarrow B\}$, where $t_i = \{I_{i,1}, I_{i,2}, \dots, I_{i,l}\}$, l is the total number of the items involved in t_i , $I_{i,j} \in I$. A and B are subsets of I , where $A \cap B = \emptyset$.

In the aforementioned example for Amazon.com, a television and a Blu-ray player are included in a set of items I . The purchasing pattern between televisions and Blu-ray players is formed as an association pattern $\{television \rightarrow Blu-ray\ player\}$. The left-side and right-side itemsets in an association pattern are called antecedent and consequent, respectively.

An association pattern is evaluated according to support and confidence. Support is an indication of how frequently an itemset occurs in given database transactions. The support value of an itemset A is computed with the number of transactions involving the itemset A in the transaction database, as follows:

$$support(A) = |t_i | A \subseteq t_i, t_i \in T| \quad (1)$$

Confidence is an indication of the reliability of an identified association pattern. The confidence of an association pattern is computed by dividing the support value of the union set of the antecedent and consequent by the support value of the antecedent in an association pattern. It is defined as follows:

$$confidence(A \rightarrow B) = \frac{support(A \cup B)}{support(A)} \quad (2)$$

The reliability of an association pattern is judged according to its confidence value. A higher confidence value indicates higher reliability of an association pattern; in contrast, a lower confidence value indicates lower reliability of an association pattern.

The concept of association rule discovery can be applied to identify change patterns between source files from a source-code repository of a software system. A source-code repository such as Git records the change history of all the source files of a software system. Ying et al. [11] and Zimmerman et al. [12] presented a change-recommendation approach based on change patterns that can be identified by mining software repositories. Given a set of source files as a query for change recommendation, the change-recommendation approach identifies the change patterns of the given source files according to the change history of the given source files. The identified change patterns that have confidence value more than the threshold are recommended as a set of change recommendations. The change-recommendation approach has a limitation for identifying meaningful change patterns for source files that have a significantly short change history or have been frequently co-changed with functionally unrelated source files [22,23]. In an experiment involving six large software systems, the change-recommendation approach found that only 25% of the change patterns were meaningful [19].

This study began to resolve the limitation of the existing change-recommendation approach. We found a solution in the co-evolution relationship between source and test files.

2.2. Coevolution of Source and Test Files

Software testing is essential to develop high quality software systems [24]. Unit tests and integrated tests are used to receive feedbacks and identify potential bugs immediately. In addition, written test code allows software project developers to understand a software system [26]. For such advantages, the software testing is widely utilized in many recent software project developments.

A source file and its corresponding test file co-evolve [32]. When a new source file is added or an existing source file is modified, their corresponding test files are created and modified. Testing comprises 30%–50% of the efforts in software project development. Previous studies [24,27,28] investigated the change impacts between source and test files. Zaidman et al. developed a visualization tool for displaying the change relations among source and test files and then analyzed the evolution history of source and test files in two open-source projects using the tool [24,27]. They showed that the evolution of the source and test files may differ according to the testing strategy employed in the software project. Marsavina et al. investigated the fine-grained code evolution of source and test files in five open-source projects using ChangeDistiller [33]. Their experiments showed that the change of a source file is often followed by the changes of its corresponding test files. In addition, they observed that closely-related source files often evolve with the same test file.

This study is inspired by the previous works. We believe that it is reasonable to consider the change history of the corresponding test file for identifying the change patterns of a source file. In our study, we focus on extracting file-level change patterns that can be obtained from white-box testing changes not module-level change pattern that can be obtained from black-box testing changes. We did not consider black-box testing changes in our approach due to the following two reasons:

- Typically, it is thought that file-level change patterns are more appropriate for assisting software project developers' tasks than module-level change patterns because a file is a basic task unit of developers in software project development.
- In a software project that has very few modules, file-level change patterns are more applicable than module-level change patterns.

3. Our Approach

In this section, we describe the overall steps of the proposed change-recommendation approach. The workflows of the proposed change-recommendation approach are briefly presented in Section 3.1, and each step is detailed in the following subsections.

3.1. Overview of Proposed Change-Recommendation Method

The basic idea of the proposed change-recommendation approach is to consider not only the change patterns of a given query source file, but also the change patterns of a test file corresponding to the given source file. Figure 1 shows the overall workflows of the proposed change-recommendation approach. The proposed change-recommendation approach consists of four steps: identifying a source–test pair, extracting commit histories, identifying change patterns, and comprising change recommendations. Given a source file as a query for a change recommendation, the proposed approach first identifies a test file corresponding to the source file using a heuristic method based on the naming convention rule between source and test files and then extracts the change histories of the source and test files by tracing the entire commit history of a Git repository. Then, the change patterns of the source and test files are identified according to the extracted change histories. Finally, a set of change recommendations is constructed using the identified change patterns.

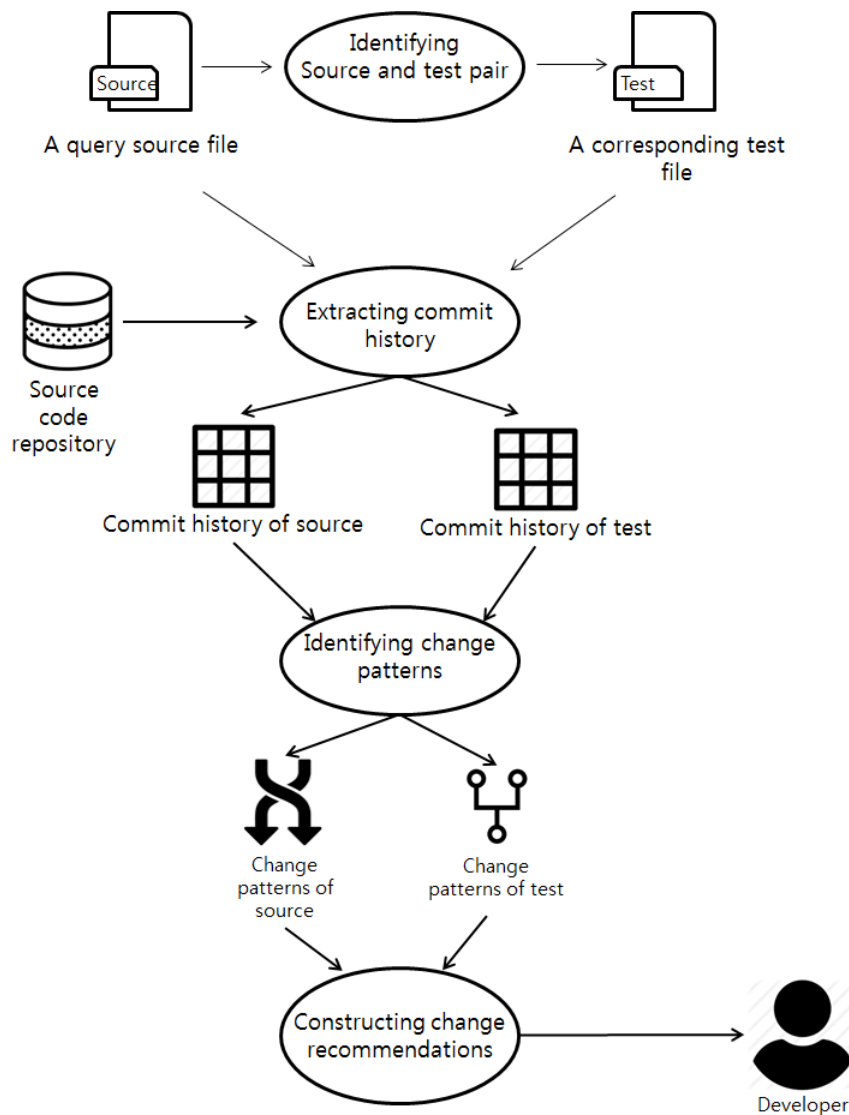


Figure 1. Workflows of the proposed change-recommendation approach.

3.2. Identifying Corresponding Test File

Software project developers typically follow a naming convention rule to identify specific test files. Under the naming convention rule, a test file is named by combining the name of its corresponding source file and the string literal “Test”. For example, a test file corresponding to a source file “Foo.java” is named with a name “FooTest.java”. The naming convention rule allows linking a source file and its corresponding test file easily. Several studies [29,30] presented a source-test traceability linking method that is based on the naming convention rule and validated the accuracy of the source-test traceability linking method.

The proposed change recommendation approach employs the source-test traceability linking method to identify a test file corresponded to a given query source file. The source-test traceability linking method considers name and path of source and test files to identify a corresponded pair of a source and test file. For example, given a source file “org/java/main/model/data/foo.java” as an input, then a test file “org/java/test/model/data/fooTest.java” is identified as the corresponded test file.

3.3. Extracting Commit Histories

Once a corresponding test file is identified, the change histories of the given query source and the corresponding test file are extracted from commit history of a source code repository of a software system such as Git repository. Git repository is a distributed version control system and is commonly used to manage changes of files in a software system.

In a Git repository, a commit history is represented by a commit metadata involving change information of several files. A commit metadata consists of a SHA-1 (Secure Hash Algorithm) hash value, committed date, commit author, a commit message and diff information. The SHA-1 hash value is an identifier for distinguishing between commits. The diff information represents several pieces of change information for changed files such as change type, previous path, new path, and textual content of changed code. Figure 2 shows an example of commit metadata represented by JSON (JavaScript Object Notation) format.

```
{
  "_id" : ObjectId("5acebd6f81c2c721c4ae96a4"),
  "project" : "jgit",
  "sha" : "d8234d310db7e730fba4e8685ac3bf8fb94046e8",
  "author" : {
    "email" : "matthias.sohn@sap.com",
    "name" : "Matthias Sohn",
    "date" : ISODate("2018-03-24T21:05:53.000Z")
  },
  "committer" : {
    "email" : "matthias.sohn@sap.com",
    "name" : "Matthias Sohn",
    "date" : ISODate("2018-03-24T23:56:53.000Z")
  },
  "message" : "Add missing @since tags for new APIWnWnThese methods were added after 4.11 so",
  "diffs" : [
    {
      "change_type" : "MODIFY",
      "old_path" : "org.eclipse.jgit/src/org/eclipse/jgit/lib/ObjectIdSerializer.java",
      "new_path" : "org.eclipse.jgit/src/org/eclipse/jgit/lib/ObjectIdSerializer.java",
      "patch" : "diff --git a/org.eclipse.jgit/src/org/eclipse/jgit/lib/ObjectIdSerializer.java b/org.eclip"
    }
  ]
}
```

Figure 2. Example of commit metadata.

For this study, we implemented a Git commit history extractor using JGit API (Application Programming Interface) [34] to extract specific commit histories from a Git repository. The JGit API is a java library implementing all the commands of the Git. Given a set of files as input, the Git commit history extractor extracts all the commit histories related to the changes of the given files by retrieving a Git repository from the most recent commit to the first commit. For each file in the given files, if a commit contains the entire path of the file in diff information of its commit metadata, the commit is extracted. The proposed change-recommendation approach uses the Git commit history extractor to extract all the commit histories related to the changes of the query source and the corresponding test file.

3.4. Identifying Change Patterns

The basic concept of the association rule discovery can be applied to change recommendation [11]. An association pattern of an itemset is interpreted as another itemset that has been frequently occurred in database transactions. Similarly, in the context of the change recommendation, a change pattern of a source file is interpreted as another source file that has been frequently co-changed in change history of the source file.

The existing change-recommendation method was developed based on the basic concept of the association rule discovery. The existing change-recommendation method analyzes all commit history of a Git repository to make a change recommendation for given source files, which may be impractical in software development because it is time-consuming to identify many unrelated change patterns

of the given source files. To avoid this shortcoming, we consider only the commit histories extracted from the previous step to identify the change patterns of the query source and corresponding test file.

To ease understanding the identification of the sets of the change patterns of the query source and corresponding test file, we first introduce a few basic definitions. Although a commit contains changes of various types of files such as source, test, text, binary file, we focus on the changes of source files in this paper. Thus, a commit is defined as a set of source files that were changed in the commit as follows:

$$c = \{f_1, f_2, \dots, f_n\} \quad (3)$$

where f_i is a source file changed in a commit c and n is the total number of the changed source files in c . For example, if source files f_1, f_2 and f_3 were changed in a commit c , the commit c is represented as a set of the source files $\{f_1, f_2, f_3\}$.

Let the commits extracted at the previous step be C , we can classify the commits in C into two sets of commits for the query source file and the corresponding test file. For example, the commits $c \in C$ that contains the query source file are classified into a set of commits for the query source file, and the commits that contains the corresponding test file are classified into a set of commits for the corresponding test file. It is defined as follows:

$$C_s = \{c | s \in c\}, C_t = \{c | t \in c\} \quad (4)$$

where s and t refer to the query source file and the corresponding test file, respectively. Based on the above definitions, the co-changed source files with the query source and the corresponding test file are defined as the change patterns of the query source and corresponding test file as follows:

$$\begin{aligned} CP_s &= \{f_i | f_i \in c_a, c_a \in C_s, s \neq f_i\}, \\ CP_t &= \{f_j | f_j \in c_b, c_b \in C_t, t \neq f_j\} \end{aligned} \quad (5)$$

where c_a and c_b are a commit involved in the C_s and C_t , respectively. CP_s and CP_t are considered to be co-changeable source files with the query source file and the corresponding test file, respectively. Figure 3 shows the overall process of identifying CP_s and CP_t .

```

1.  $CP_s \leftarrow \emptyset$ 
2.  $CP_t \leftarrow \emptyset$ 
3. for  $c_a$  in  $C_s$  do
4.   for  $f_i$  in  $c_a$  do
5.     if  $f_i \neq s$ 
6.        $CP_s \leftarrow CP_s \cup f_i$ 
7.   end for
8. end for
9. for  $c_b$  in  $C_t$  do
10.  for  $f_j$  in  $c_b$  do
11.    if  $f_j \neq t$ 
12.       $CP_t \leftarrow CP_t \cup f_j$ 
13.    end for
14. end for

```

Figure 3. Process of identifying of change patterns of a pair of source and test files.

3.5. Constructing Change-Recommendation Set

A change-recommendation set for the given source file is constructed according to the CP_s and CP_t identified in the previous step. The proposed change-recommendation approach first selects k change patterns from CP_s in the order of their confidence values and includes the selected change patterns in a change-recommendation set. It then adds all the change patterns in CP_t to the change-recommendation set. Similar to association patterns, the confidence of a change pattern is determined by the co-change frequency of the source files involved in a change pattern. For a change pattern $\{s \rightarrow f_i\}$ in CR_s ,

the co-change frequency of the given source file s and the other source file f_i is the number of commit histories in C_s that involve the source file f_i . It is defined as follows:

$$co-change_{s,f_i} = |\{c_j | f_i \in c_j, c_j \in C_s\}| \quad (6)$$

For example, if a source file f_1 is contained in three commits c_1, c_2 , and $c_3 \in C_s$, $co-change_{s,f_1}$ is computed by 3 ($|\{c_1, c_2, c_3\}|$). Using Equation (6), the confidence of the change pattern $\{s \rightarrow f_i\}$ is computed by dividing the co-change frequency of s and f_i by the number of commits in the entire commit history of the query source file s , as follows:

$$CP_{s \rightarrow f_i} = \frac{co-change_{s,f_i}}{|C_s|} \quad (7)$$

For example, if C_s has five commits $\{c_1, c_2, c_3, c_4, c_5\}$, the confidence of a change pattern $CP_{s \rightarrow f_1}$ is computed by 0.6. Using the Equation (7), the confidences of all the change patterns in CP_s are computed and a set of change recommendations is constructed by selecting the top k change patterns from CP_s in the order of the confidence values. Then, all the change patterns in CP_t are added to the change-recommendation set. Finally, the change-recommendation set for the given source file s is determined as follows:

$$ChgRec_s = CP_{s,k} \cup CP_t \quad (8)$$

Here, $CP_{s,k}$ comprises the k change patterns selected from CP_s . The value of k can be arbitrarily chosen. If the value of k is high, meaningless change patterns may be recommended, on the other hand, if the value of k is extremely low, meaningful change patterns may be missing. Therefore, the value of k should be determined between 10 and 30. For example, change patterns of the given query source file and the corresponding test file are given as $CP_s = \{f_{1,0.8}, f_{2,0.8}, f_{3,0.6}, f_{4,0.5}, f_{5,0.2}\}$, where $f_{1,0.8}$ is abbreviation of $CP_{s \rightarrow f} = 0.8$, and $CP_t = \{f_6, f_7\}$, respectively, and the value of k is chosen by three, $CP_{s,k}$ is determined as $\{f_1, f_2, f_3\}$ and then $ChgRec_s$ is determined as $\{f_1, f_2, f_3, f_4, f_5\}$.

$ChgRec_s$ involves not only the change patterns of the given query source file but also the change patterns of its corresponding test file. Although the proposed change-recommendation approach cannot identify any change patterns from $CP_{s,k}$ owing to the short change history of the given source file, it can recommend alternative change patterns from CP_t .

4. Experiment

In this section, we report the results of an experiment performed to evaluate the performance of the proposed change-recommendation approach. The objective of the experiment is to investigate whether the proposed change-recommendation approach has higher performance than the existing change-recommendation approach. The data used for the experiment and the experiment settings are described in Sections 4.1 and 4.2, respectively. The metric for evaluating the performance of the change-recommendation approaches is introduced in Section 4.3. The experimental results are presented in Section 4.4.

4.1. Experimental Data

For this experiment, the several software projects that employ at least one test framework in their development and allow public access to their Git repository are required. We chose the following projects for the experiment: commons-lang [35], commons-math [36], JGit [34], Maven [37], Flink [38], and Wicket [39]. Commons-lang and commons-math are utility libraries for Java application project development; JGit is a Java library implementing the commands of the Git; Maven is a tool for software project management and integration; Flink is an open-source framework for stream processing; and Wicket is an open-source web application framework based on components. These projects are

developed using the JUnit test framework [40] and allow access to their Git repository. Hence, we selected these open-source projects for the experiment.

To collect the experimental dataset from the aforementioned open source projects for our change recommendation experiment, we first cloned the Git repositories of the open source projects from GitHub [41]. We then extracted the commits that contain the changes of source files or test files from the cloned Git repositories. In the cloned Git repositories of the open source projects, we determined the pairs of the source and test files that can be used as the queries for the change recommendations in the experiment. First, we identified the corresponding source and test files using the source-test linking method mentioned in Section 3.2. We then excluded the pair of the source and test files that have insufficient commit histories to identify their change patterns. The pairs of the source and test files with less than five commits were excluded from the identified source-test pairs. Table 1 shows the experimental dataset collected from the open source projects. For each open source project, the column of #. Commits refers to the number of commits submitted by developers within the Commit period. The columns of #. Changed Source Files and #. Changed Test Files refer to the numbers of source and test files that have been changed in the number of commits (#.Commits), respectively. The column of #. Pairs of Source and Test refers to the number of paired source and test files in the changed source and test files (#. Changed Source Files and #. Changed Test Files). The column of #. Related Commits refers to the number of commits in which the source files or the test files has been changed. The dataset in Table 1 varies across the open source projects. This is because the open source projects are different in development period, functionality implemented, and testing strategy that they adopted.

Table 1. Experimental datasets.

Project	Commit Period	#. Commits	#. Changed Source Files	#. Changed Test Files	#. Pairs of Source and Test	#. Related Commits
commons-lang	2002-07-19 ~ 2018-03-11	5632	320	452	101	234
commons-math	2003-05-13 ~ 2018-03-18	7231	2954	1735	510	501
Jgit	2009-09-30 ~ 2018-03-28	5836	1037	437	192	1490
Maven	2003-09-02 ~ 2018-03-21	12,218	767	925	49	442
Flink	2010-12-16 ~ 2018-07-25	17,289	754	396	78	173
Wicket	2004-09-22 ~ 2018-07-25	32,366	1401	2455	167	659

4.2. Experimental Setting

In the experiment, we compare the proposed approach with the existing method developed by Ying et. al [11]. The existing method is based on the association rule mining. Given a query source file and a number of change patterns to be recommended (k) as input, the existing method first extracts the commits of the given query source file from a source code repository. Change patterns are then formed by applying association rule mining algorithm. Finally, k change patterns are recommended according to the confidence value of the change patterns.

To make a change recommendation, the proposed and existing change-recommendation approaches require several arguments, such as a query source file, training commits, and a number of recommended patterns, k , from a set of the change patterns of the query source file. In this experiment, we used the source files involved in the identified source–test pairs as the query source files and used all of the commits related to the source-test pairs in each project for training and evaluation. The proposed approach usually employs more commits for training than the existing approach. For a source–test pair and an evaluation commit, the existing approach uses the commits that precede the evaluation commit in a set of commits of the source for training, while the proposed approach uses the commits that precede the evaluation commit in both sets of the commits of the source and test. For example, given a source–test pair $\langle s, t \rangle$, their commit history $\{c_{s,1}, c_{t,1}, c_{s,2}, c_{t,2}, c_{s,3}\}$ and an evaluation commit $c_{s,2}$, the existing approach and proposed approach choose the commits $\{c_{s,1}, c_{s,2}\}$ and $\{c_{s,1}, c_{t,1}, c_{s,2}, c_{t,2}\}$ for training, respectively. We set the value of k to 10 according to the previous work related to recommendation system [42]. Thus, for each change recommendation, the existing approach makes only 10 change recommendations from the change patterns of a query

source file, while the proposed approach makes 10 change recommendations and additional change recommendations from the change patterns of a query source file and the corresponding test file.

4.3. Evaluation Metric

To evaluate the performance of the proposed change-recommendation approach, we used an accuracy measurement method. Accuracy measurement is widely used to evaluate various information retrieval methods and recommendation systems [20,42–46]. In this study, the accuracy of a change recommendation is computed as follows:

$$Accuracy = \frac{|ChgRec_s|}{|AC_s|} \times 100\% \quad (9)$$

where AC_s is a set of actually co-changed source files with a given query source file s . The accuracy ranges from 0 to 1. If none of the recommended source files is included in the set of actually-changed source files, the accuracy is 0. In contrast, if all the recommended source files are involved in a set of actually-changed source files, the accuracy is 1. For example, in a change recommendation, given a set of recommended source files $CR_{s,5} = \{f_a, f_c, f_e, f_f, f_g\}$ and a set of actually-changed source files $AC_s = \{f_a, f_b, f_c, f_d, f_e\}$, the accuracy for the change recommendation is 60%.

4.4. Result

Table 2 shows the average accuracy of the proposed and existing approaches in the experiment projects. For all the projects, the proposed approach obtained a significantly higher average accuracy than the existing approach. For the projects, as listed in Table 2, the proposed approach obtained average accuracies of 82%, 70%, 58%, 48%, 56%, and 50%, while the existing approach obtained average accuracies of 20%, 13%, 11%, 27%, 16%, and 16%. On average, the accuracy of the proposed was improved by 43% compared with the existing approach. Comparing the average accuracy between the proposed and existing approaches for each project reveals that the proposed approach improved the accuracy by 62%, 56%, 47%, 21%, 40%, and 34%.

To find out how the proposed approach obtains the improved results, we investigated the change-recommendation results of the proposed and existing approaches. Through the investigation, we observed that even for query source files that have relatively few commit histories, the proposed approach can make correct change patterns from the commit histories of corresponding test files, while the existing approach cannot identify any change patterns.

Table 2. Results for the accuracy and improvements of the proposed approach.

Project	Accuracy		
	Existing	Proposed	Improvement
commons-lang	20%	82%	62%
commons-math	13%	70%	56%
JGit	11%	58%	47%
Maven	27%	48%	21%
Flink	16%	56%	40%
Wicket	16%	50%	34%
Avg.	17%	61%	43%

We summarized the root-causes of the incorrect change pattern identification. Figure 4 shows the four types of categories of the root-causes, such as Development environment, Project, Testing and Commit activity. ‘Short development period’ categorized in ‘Development environment’ is a major cause of lacking commit history to identify change patterns. ‘Absence of guide’ and ‘Absence of

manual' categorized in 'Project' may cause mistakes of developers in committing. The mistakes may be major causes on 'Missed commit', 'Overlapped commit', and 'Delayed commit'. 'Absence of testing strategy' and 'Absence of testing framework' may also affect the quality of commit history.

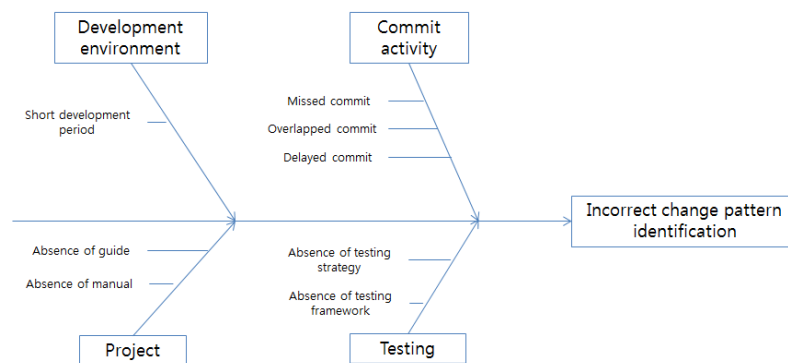


Figure 4. Fishbone diagram for root-causes of Incorrect change pattern identification.

Furthermore, we performed a pairwise t-test statistical analysis to evaluate the difference of the results. A null hypothesis and its alternative hypothesis for the statistical analysis are presented as follows.

H_{null} : There are no statistically significant differences between the proposed and existing approaches.

$H_{alternative}$: There are statistically significant differences between the proposed and existing approaches.

We performed a pairwise t-test for all of the paired recommendation results by using the student's t-test function in the R package. Table 3 shows the p -values obtained in the statistical analysis. For all of the experiment projects, the p -values are less than 0.01. It means that the null hypothesis is rejected with 99% confidence. Thus, it is shown that there are statistically significant differences between the proposed and existing approaches in accuracy. This means that the average accuracy obtained by the proposed approach is statistically better than the average accuracy obtained by the existing approach.

Table 3. Results of the statistical t -test.

Project	p -Value	H_{null}
commons-lang	$<2.2 \times 10^{-16}$	Reject
commons-math	$<2.2 \times 10^{-16}$	Reject
JGit	$<2.2 \times 10^{-16}$	Reject
Maven	$<2.2 \times 10^{-16}$	Reject
Flink	$<2.2 \times 10^{-16}$	Reject
Wicket	$<2.2 \times 10^{-16}$	Reject

5. Discussion, Implications, Limitations, and Conclusion

5.1. Discussion and Implications

The experimental results shown in Section 4.4 demonstrate that the proposed approach can obtain better performance than the existing method. In this section, we discuss why the proposed approach can achieve the performance. The significant difference between the proposed approach and the existing method is Equation (8). The proposed approach contains the change patterns of the query source and corresponding test file while the existing method contains only the change patterns of the query source file into a set of change recommendations. The benefit of the difference is that the change history of the corresponding test file can be considered to discover appropriate change patterns of the

query source file when a query source file has short change history. We believe that the advantage of the proposed approach can complement the limitation of the existing method.

This study is impressed by the nature of evolution of test files revealed in previous studies [24,26–28,31,47]. In general, source and test files co-evolve in software project development. Modifications of a source file affect several related test files. A test file is needed to verify several related source files. Thus, the change history of a test file reflects the changes of source files that are explicitly related to each other. Hence, we consider that when identifying change patterns of a source file, it is reasonable to consider the change history of the test file explicitly related to the source file. The experimental results in Section 4.4 emphasize that change history of test files should be considered when identifying change patterns of corresponding source files. We believe that this study can progress further studies on change recommendations, and also that the proposed approach in this study can significantly contribute to real-world software project development, especially for young software projects where most source files have a short commit history.

5.2. Limitation

The internal validity of this study is related to the identification of source and test pairs. The proposed approach requires the change history of a test file that is related to a given source file. If an incorrect test file is identified, the change patterns obtained from the identified test file may be unreliable. Source–test traceability linking is an important research field. An approach that can perfectly identify traceability links between source and test files has not yet been reported. In this study, we used a source–test traceability linking approach based on the naming convention rule between source and test files to identify test files corresponding to given source files. Then, we manually verified whether the identified pairs were correct in the experiment. Typically, software project developers write test files by following the naming convention rule in most software projects [29,30]. Therefore, in this study, it is believed to minimize the bias by using the source–test traceability linking approach in this study.

The external validity of this study is related to the generalization of the experimental results. As the experimental results are obtained from six open-source projects, it cannot be generalized to all software projects. Therefore, it is required to conduct additional experiments on various software projects to reduce the bias. However, the experiment projects have different scales and involve different domains. Thus, we believe that the bias is reduced.

5.3. Conclusion

In this study, we proposed a novel change-recommendation approach that considers not only the change history of a given source file but also the change history of a test file corresponding to the given source file. Given a source file as a query, the proposed approach identifies a corresponding test file by using a source–test linking approach based on the naming convention rule between source and test files and extracts the commit histories of the given source file and the identified test file from a Git repository. Then, the change patterns of the source and test files are identified according to the extracted commit histories. Finally, a set of change recommendations is constructed using the identified change patterns of the source and the test files. The proposed change-recommendation approach is evaluated for six open-source projects. For the open-source projects, the proposed change-recommendation approach obtained significantly better accuracy than the existing change-recommendation approach.

In future works, we plan to conduct additional experiments with various software projects to reduce the bias. In addition, we will study to employ our approach for functional change recommendation. We believe that the proposed change recommendation approach can be extended to make functional change recommendations. For instance, if changes of source files are classified into functional changes of a software system, the functional change recommendations can be made by considering functional change history and corresponding testing history of a software system.

Author Contributions: J.K. collected all of the experimental data, performed the experiment, analyzed the result of the experiments, and wrote the paper. E.L. supervised this study, suggested the research approach, and designed the overall experiment method.

Acknowledgments: This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2016R1D1A1B03931276). This study was also supported by the BK21 Plus Project (SW Human Resource Development Program for Supporting Smart Life), funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (21A20131600005).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Lehman, M.M. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.* **1979**, *1*, 213–221. [[CrossRef](#)]
2. Gall, H.; Hajek, K.; Jazayeri, M. Detection of Logical Coupling Based on Product Release History. In Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, 16–19 March 1998; pp. 190–198.
3. Fluri, B.; Gall, C.H.; Pinzger, M. Fine-Grained Analysis of Change Couplings. In Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation, Budapest, Hungary, 30 September–1 October 2005; pp. 66–74.
4. Poshyvanyk, D.; Marcus, A. The Conceptual Coupling Metrics for Object-Oriented Systems. In Proceedings of the 22nd IEEE International Conference on Software Maintenance, Philadelphia, PA, USA, 24–27 September 2006; pp. 469–478.
5. Poshyvanyk, D.; Marcus, A.; Ferenc, R.; Gyimothy, T. Using information retrieval based coupling measures for impact analysis. *Empir. Softw. Eng.* **2009**, *14*, 5–32. [[CrossRef](#)]
6. D'Ambros, M.; Lanza, M.; Robbes, R. On the Relationship Between Change Coupling and Software Defects. In Proceedings of the 16th Working Conference on Reverse Engineering, Lille, France, 13–16 October 2009; pp. 135–144.
7. Mondal, M.; Roy, K.C.; Schneider, A.K. Insight into a Method Co-change Pattern to Identify Highly Coupled Methods: An Empirical Study. In Proceedings of the 21st IEEE International Conference on Program Comprehension, San Francisco, CA, USA, 20–21 May 2013; pp. 103–112.
8. Rahman, S.M.; Roy, K.C.A. Change-Type Based Empirical Study on the Stability of Cloned Code. In Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, Canada, 28–29 September 2014; pp. 31–40.
9. Canfora, G.; Cerulo, L. Impact Analysis by Mining Software and Change Request Repositories. In Proceedings of the 11th IEEE International Software Metrics Symposium, Washington, DC, USA, 19–22 September 2005; pp. 29–37.
10. Zanjani, M.B.; Swartzendruber, G.; Kagdi, H. Impact analysis of change requests on source code based on interaction and commit histories. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 162–171.
11. Ying, T.T.A.; Murphy, C.G.; Ng, R.; Chu-Carroll, C.M. Predicting Source Code Changes by Mining Change History. *IEEE Trans. Softw. Eng.* **2004**, *30*, 574–586. [[CrossRef](#)]
12. Zimmermann, T.; Weißgerber, P.; Diehl, S.; Zeller, A. Mining Version Histories to Guide Software Changes. *IEEE Trans. Softw. Eng.* **2005**, *31*, 429–445. [[CrossRef](#)]
13. Dondero, M.R. Predicting Software Change Coupling. Ph.D. Thesis, The Faculty of Drexel University, Philadelphia, PA, USA, 2008.
14. Robbes, R.; Pollet, D.; Lanza, M. Logical Coupling Based on Fine-Grained Change Information. In Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, 15–18 October 2008; pp. 42–46.
15. Canfora, G.; Ceccarelli, M.; Cerulo, L.; Penta, D.M. Using Multivariate Time Series and Association Rules to Detect Logical Change Coupling: An Empirical Study. In Proceedings of the IEEE International Conference on Software Maintenance, Timisoara, Romania, 12–18 September 2010; pp. 1–10.

16. Agrawal, R.; Imieliński, T.; Swami, A. Mining association rules between sets of items in large databases. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, 25–28 May 1993; pp. 207–216.
17. Tan, P.N.; Steinbach, M.; Kumar, V. *Introduction to Data Mining*; Pearson Education: Boston, MA, USA, 2006.
18. Siddiqui, T.; Ahmad, A. Data Mining Tools and Techniques for Mining Software Repositories: A Systematic Review. In *Big Data Analytics*; Aggarwal, V., Bhatnagar, V., Mishra, D., Eds.; Springer: Singapore, 2018; Volume 654, pp. 717–726.
19. Rolfsnes, T.; Alesio, D.S.; Behjati, R.; Moonen, L.; Binkley, W.D. Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis. In Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Suita, Japan, 14–18 March 2016; pp. 201–212.
20. Rolfsnes, T.; Moonen, L.; Alesio, D.S.; Behjati, R.; Binkley, D. Improving Change Recommendation using Aggregated Association Rules. In Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories, Austin, TX, USA, 14–15 May 2016; pp. 73–84.
21. Rolfsnes, T.; Moonen, L.; Di Alesio, S.; Behjati, R.; Binkley, D. Aggregating Association Rules to Improve Change Recommendation. *Empir. Softw. Eng.* **2018**, *23*, 987–1035. [[CrossRef](#)]
22. Herzig, K.; Zeller, A. The impact of tangled code changes. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 121–130.
23. Kirinuki, H.; Higo, Y.; Hotta, K.; Kusumoto, S. Hey! are you committing tangled changes? In Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, India, 2–3 June 2014; pp. 262–265.
24. Zaidman, A.; van Rompaey, B.; Demeyer, S.; van Deursen, A. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, Norway, 9–11 April 2008; pp. 220–229.
25. van Rompaey, B.; Zaidman, A.; van Deursen, A.; Demeyer, S. *Comparing the Co-Evolution of Production and Test Code in Open Source and Industrial Developer Test Processes through Repository Mining*; Technical Report TUD-SERG-2008-034; Delft University of Technology, Software Engineering Research Group: Delft, The Netherlands, 2008.
26. Lubsen, Z.; Zaidman, A.; Pinzger, M. Using Association Rules to Study the Co-evolution of Production & Test Code. In Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, Vancouver, BC, Canada, 16–17 May 2009; pp. 151–154.
27. Zaidman, A.; van Rompaey, B.; van Deursen, A.; Demeyer, S. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empir. Softw. Eng.* **2011**, *16*, 325–364. [[CrossRef](#)]
28. Marsavina, C. Studying Fine-Grained Co-Evolution Patterns of Production and Test Code. Master's Thesis, Delft University of Technology, Delft, The Netherlands, 2014.
29. van Rompaey, B.; Demeyer, S. Establishing Traceability Links between Unit Test Cases and Units under Test. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 24–27 March 2009; pp. 209–218.
30. Qusef, A.; Bavota, G.; Oliveto, R.; De Lucia, A.; Binkley, D. SCOTCH: Test-to-Code Traceability using Slicing and Conceptual Coupling. In Proceedings of the 27th IEEE International Conference on Software Maintenance, Williamsburg, VA, USA, 25–30 September 2011; pp. 63–72.
31. Nguyen, L.; Vo, B.; Nguyen, L.; Fournier-Viger, P.; Selamat, A. ETARM: An efficient top-k association rule mining algorithm. *Appl. Intell.* **2018**, *48*, 1148–1160. [[CrossRef](#)]
32. Peng, M.; Sundararajan, V.; Williamson, T.; Minty, P.E.; Smith, C.T.; Doktorchik, T.A.C.; Quan, H. Exploration of association rule mining for coding consistency and completeness assessment in inpatient administrative health data. *J. Biomed. Inform.* **2018**, *79*, 41–47. [[CrossRef](#)] [[PubMed](#)]
33. Fluri, B.; Gall, C.H. Classifying Change Types for Qualifying Change Couplings. In Proceedings of the International Conference on Program Comprehension, Athens, Greece, 14–16 June 2006; pp. 35–45.
34. JGit—The Eclipse Foundation. Available online: <https://www.eclipse.org/jgit/> (accessed on 19 October 2018).
35. Lang—Home. Available online: <https://commons.apache.org/proper/commons-lang/> (accessed on 19 October 2018).

36. Math—Commons Math: The Apache Commons Mathematics Library. Available online: <http://commons.apache.org/proper/commons-math/> (accessed on 19 October 2018).
37. Maven—Welcome to Apache Maven. Available online: <https://maven.apache.org/> (accessed on 19 October 2018).
38. Apache Flink: Stateful Computations over Data Streams. Available online: <https://flink.apache.org/> (accessed on 19 October 2018).
39. Apache Wicket. Available online: <https://wicket.apache.org/> (accessed on 19 October 2018).
40. JUnit 5. Available online: <https://junit.org/junit5/> (accessed on 19 October 2018).
41. The World's Leading Software Development Platform—GitHub. Available online: <https://www.github.com> (accessed on 19 October 2018).
42. Kim, J.; Lee, E. Understanding Review Expertise of Developers: A Reviewer Recommendation Approach Based on Latent Dirichlet Allocation. *Symmetry* **2018**, *10*, 114. [[CrossRef](#)]
43. Xia, X.; Loy, D.; Wang, X.; Zhou, B. Tag recommendation in software information sites. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 287–296.
44. Marung, U.; Theera-Umpon, N.; Auephanwiriyakul, S. Top-N Recommender Systems Using Genetic Algorithm-Based Visual-Clustering Methods. *Symmetry* **2016**, *8*, 54. [[CrossRef](#)]
45. Guo, Y.; Wang, M.; Li, X. An Interactive Personalized Recommendation System Using the Hybrid Algorithm Model. *Symmetry* **2017**, *9*, 216. [[CrossRef](#)]
46. Zheg, J.; Li, D.; Arun Kumar, S. Group User Profile Modeling Based on Neural Word Embeddings in Social Networks. *Symmetry* **2018**, *10*, 435. [[CrossRef](#)]
47. Lamkanfi, A.; Demeyer, S. Studying the Co-evolution of Application Code and Test Cases. In Proceedings of the 9th Belgian-Netherlands Software Evolution Seminar (BENEVOL 2010), Lille, France, 16 December 2010.



© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).