Refactoring Edit History of Source Code

Shinpei Hayashi*, Takayuki Omori†, Teruyoshi Zenmyo*, Katsuhisa Maruyama†, and Motoshi Saeki*
*Tokyo Institute of Technology, Tokyo 152–8552, Japan Email: {hayashi,zenmyo,saeki}@se.cs.titech.ac.jp
†Ritsumeikan University, Shiga 525–8577, Japan Email: {maru@,takayuki@fse.}cs.ritsumei.ac.jp

***Abstract*—This paper proposes a concept for refactoring an edit history of source code and a technique for its automation. The aim of our history refactoring is to improve the clarity and usefulness of the history without changing its overall effect. We have defined primitive history refactorings including their preconditions and procedures, and large refactorings composed of these primitives. Moreover, we have implemented a supporting tool that automates the application of history refactorings in the middle of a source code editing process. Our tool enables developers to pursue some useful applications using history refactorings such as task level commit from an entangled edit history and selective undo of past edit operations.**

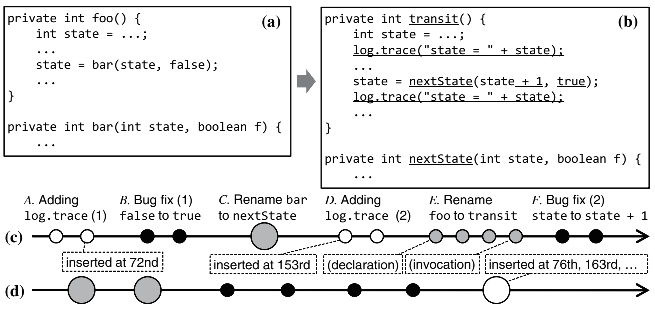
***Keywords*-refactoring, edit history, software configuration management.**

I. Introduction

Managing the recorded edit history of source code serves an important role in enhancing software development productivity. The simplest example is that developers can undo/redo their edits by recording an edit history. In addition, some approaches that replay edit operations conducted by developers in the past have been attempts to support program comprehension activities [1], [2].

Well-managed edit histories help developers considerably. For instance in software configuration management (SCM), a well-known policy named *task level commit* suggests that developers not commit changes related to more than two tasks to a SCM repository [3]. In accordance with the task level commit, efforts of each task can be adopted or reverted flexibly. Then we can manage the changes intuitively with comprehensible granularity. In addition, several open source software projects have adopted a patch-based flow. To realize the submitted patches to be accepted, developers are forced to make the patches readily comprehensible for patch reviewers. To develop patches for such projects, edit histories should be managed so that the patches comprise only related edits.

However, in actual software development, keeping edit histories well-managed is not a trivial task. Our previous study [4] reported that mixed changesets could reduce reuse, revert, and understanding of past changes. Developers often make changes of various kinds in a single term. For instance, a developer who notices a design issue in fixing some bugs would merge edits for refactoring and bug fixes [5], [6]. In such circumstances, changes corresponding with



```
private int foo() {
    int state = ...;
    ...
    state = bar(state, false);
    ...
}

private int bar(int state, boolean f) {
    ...
```

```
private int transit() {
    int state = ...;
    log.trace("state = " + state);
    ...
    state = nextState(state + 1, true);
    log.trace("state = " + state);
    ...
}

private int nextState(int state, boolean f) {
    ...
```

Figure 1. Example of an entangled edit history and refactoring of it.

multiple tasks are often performed at once in a working copy. Consequently, the difference based on the obtained edits does not adapt to the task level commit. They must be restructured.

Fig. 1 presents an example of an entangled history. In this example situation, a developer sought to fix a bug included in the method `foo` as well as to rename some methods. The developer edited the source code shown in (a) and obtained that shown in (b). Underlining shown in (b) highlights the modified portions. Arrow (c) represents the sequence of the obtained edit history. The developer first inserted logging method invocations (loggings) `log.trace(···)` and executed the program to observe their behavior (changes A shown in Fig. 1). Next, the developer changed the second parameter of the invocation of `bar` (B). Additionally, the developer renamed `bar` to `nextState` to improve its comprehensibility using the automated support of the refactoring browser (C). This change is shown in a larger circle because it corresponds with multiple edit portions. The developer also added loggings (D), renamed `foo` to `transit` manually (E), and finally modified the first parameter of `nextState` as the last change (F).

The resulting history of these edits is difficult to understand because it reflects directly on the trial-and-error process pursued by the developer. Problems in the history are summarized as follows. First, the history contains multiple changes according to multiple tasks, such as bug fixes, loggings, and refactorings. Second, the rename refactoring of `foo` (E) was performed manually without using the refactoring browser. Consequently, the resulting changes obtained using this refactoring were separated and distributed throughout the history. Third, loggings should be removed if these are not used for future development, based on the development policy of the target project. However, the changes related to the loggings are distributed over the history. They are difficult to revoke automatically while maintaining other changes. This example indicates a gap separating the structures of a history suitable for understanding and for use and that the structures recorded during practical software development. Bridging this gap would contribute to productivity enhancement of software development processes.

To bridge this gap, this paper proposes a new technique of *edit history refactoring* inspired by code refactoring[1]. The edit history shown in Fig. 1(d) is the refactored version to solve the problems above. First, code refactorings, bug fixes, and insertion of loggings are placed continuously. Therefore, we can easily commit each type of change to the underlying SCM repository. Next, the changes related to the refactoring performed manually are now merged into a single change. Additionally, the insertions of loggings are placed as if these changes are performed after refactorings and bug fixes. Therefore, developers easily undo this change if they regard these changes as unnecessary.

We have implemented a prototype tool for automating the technique. Using this tool, users can perform history refactorings to an edit history obtained from the Eclipse code editor. Additionally, several applications such as task level commit from an entangled edit history or selective undo of past edit operations are available using this tool.

II. Edit History Refactoring

A. Overview

A history refactoring is defined as a restructuring of an edit history for improving the usability and/or understandability of the history without changing its overall effect. Therefore, the application result of all edits in the original history must be equal to that of all edits in the refactored history for the same input source code.

Through history refactorings, the offsets of some edits are modified for preservation of the consistency of the whole history. For example, in Fig. 1(d) the change inserting a logging into the 72nd position of source code is rewritten to the insertion into the 76th position.

One important concept in our definition of edit histories is *grouping*. Similarly to source code refactorings, the configurations of history refactorings are sometimes complicated and time consuming because history refactorings rewrite data on time series. To produce configurations of history refactorings easily, our definitions include grouping of changes such as "code refactorings" and "bug fixes" in the example given above (denoted by the colors in Figs. 1(c)–1(d)); history refactorings are configurable based on the groups.

One challenge encountered in defining history refactoring operations is how to guarantee that each operation preserves the overall effect of the target edit history. To solve this, we have followed a refactoring manner: we first define primitive small refactorings accurately; then we define larger refactorings as compositions of the pre-defined primitives. We can regard that the composed refactorings also preserve the characteristic if each of primitives is guaranteed to preserve a characteristic of an edit history. This approach frees us from developing complex pre-conditions for large history refactorings.

B. Preliminary Definitions

In this paper, we call a series of addition, removal, or replacement of a source code fragment a *chunk*. A chunk $h := (t, f, o, r, a)$ is a tuple representing modification of the characters on a source code file f, where t is the date when the edit was performed, o is the starting offset of the edit, r is the removed string from the pre-modified file, and a is the added string to the post-modified file. When h is a pure addition or pure removal, r or a is blank respectively. The delta of the whole source code length affected by h is defined as $\text{len}(h) := a.\text{length} - r.\text{length}$. Each element of an edit history $H := c_1 c_2 \cdots c_N$ is a *change*. A change is a pair of a sequence of chunks $h_1 h_2 \cdots h_n$ and a group $g \in G$ to which the change belongs: $c := (h_1 h_2 \cdots h_n, g)$.

Changes are constructed according to the operations that developers have actually performed on their terminal. For example, according to an edit of source code by manual keystrokes, a change including only a single chunk of insertion is constructed and then added to the edit history. In contrast, according to several large operations such as a built-in replace command or code refactorings using the refactoring browser corresponding to multiple edits of source code, a large change including multiple chunks that replace various portions is constructed.

History refactorings add, remove, or restructure the changes in an edit history while preserving (1) the whole effect of all chunks included in the history and (2) the added and removed strings of a chunk. Considering a situation in which we have an edit history H, we can obtain S' when we apply all changes in H to source code S in its order. Presuming that H' is a refactored edit history of H, then the following apply. Condition 1 means that we can obtain the same result S' when we apply all changes in H' to the same original source code S in its order. Condition 2 means that history refactorings regard chunks as atomic elements and do not change their internal structure (except for their offsets). This condition avoids splitting an edit performed using a developer at once into multiple chunks because

[1]In this paper, refactoring targets of two types are discussed: source code and edit histories. For clarity, we respectively call these *code refactoring* and *history refactoring* except for the cases for which the discussed type is clear.

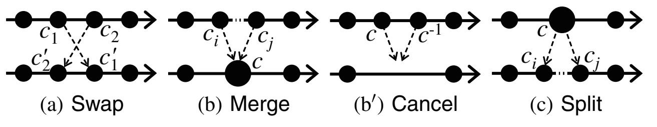


Figure 2. Primitive history refactorings.

history refactorings specifically examine the organization of chunks.

C. Primitive Refactorings

We have defined four primitive history refactoring operations. Figs. 2(a)–(d) show overviews of these primitive refactorings. In these figures, solid arrows and circles respectively represent histories and changes. For each figure, the bottom history is a refactored version of the top history. Swap swaps the execution order of target changes. Merge composes multiple changes into a single change. Cancel removes a change together with its adjacent inverse. Split splits a change into multiple changes in terms of a specific perspective (such as their types: classes or methods). Cancel is defined as a special version of Merge when the merged result does not affect source code; then we erase the resulting empty change.

As we did also for code refactorings, we could define pre-conditions, post-conditions, and transformation procedures for history refactorings. The pre-condition of Swap in short is that the right-hand-side change c_2 is independent of the left-hand-side change c_1. The pre-condition of Merge is that the changes to be merged are adjacent and belong to the same group. Additionally, we can naturally define the post-condition for all history refactorings as the invariance of the overall effect of the target edit history.

Swap is defined based on the commutation of chunks. This operation assumes that two chunks $h_i = (t_i, f_i, o_i, r_i, a_i)$ and $h_j = (t_j, f_j, o_j, r_j, a_j)$ are performed in this order, and rewrites the offsets of these chunks as if they are executed in the reverse order; h_i is executed immediately after the execution of h_j. The offsets are modified as follows:

$$(o'_j, o'_i) = \begin{cases} (o_j - \mathrm{len}(h_i), o_i) & \textit{if } o_i \geq o_j, \\ (o_j, o_i + \mathrm{len}(h_j)) & \textit{otherwise} \end{cases}$$

The pre-condition of Swap is defined based on the commutability of the related chunks. The commutation does nothing and succeeds if the target files of the given chunks differ ($f_i \neq f_j$). Otherwise, a chunk cannot be commuted if the chunks after commutation violate Condition 2 of history refactorings shown in Section II-B. For example, commutations fail when the removed range includes an added range because the target of the removal will vanish through commutation.

D. Large Refactorings

By composing multiple refactorings, *large refactoring* is definable. An overview of the defined large refactorings is shown in Fig. 3. Reorder reorders the sequence of changes from an entangled edit history according to their belonging groups [4]. Reconfigure extracts, from a large change, some edits corresponding with focused other changes.

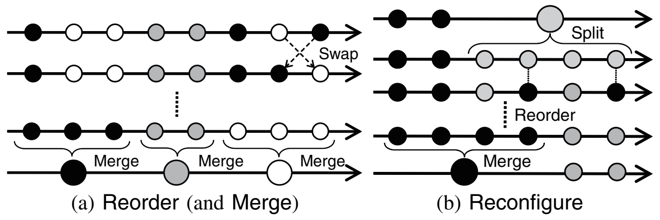


Figure 3. Large history refactorings.

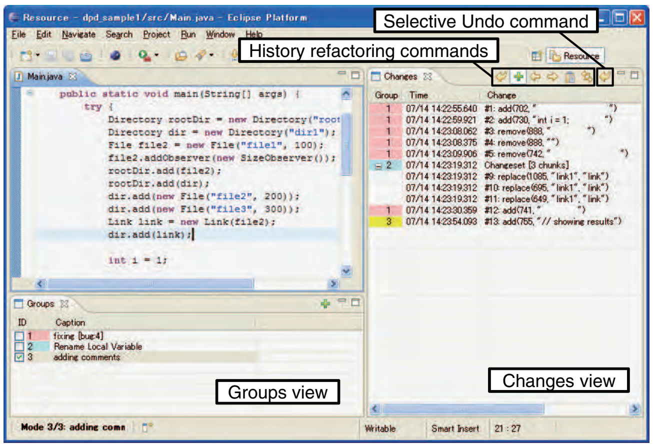
In Reorder, first we apply Swap again and make the changes belonging to the same group adjacent by sorting them. The criterion for the sorting is the given group order representing how the changes should be ordered. Finally, the changes for each group are merged into a single change using Merge if necessary. Whether all Swap operations succeed or not depends on the given order of groups. The pre-condition of the given group order is based on our reordering technique [4]. Developers can also use a group order that has been decided automatically based on the commutability of chunks performed in the past.

Reconfigure is used when users want to limit the effect of a large change according to other changes. Some examples of the target large changes are code refactorings and formatting. In Reconfigure, first we split the focused large change into multiple changes using Split. Next, we make changes that share the editing targets with the existing changes belonging to a new group. Additionally, these changes are moved to make them adjacent with the existing changes using Reorder. Finally, the changes belonging to the focused group are merged using Merge.

III. Supporting Tool

We have implemented Historef—a supporting tool for automating history refactorings in Java development. Historef is designed as a plug-in of Eclipse and is collaborating with OperationRecorder [7] which collects the edit operations that the owner developer of Eclipse actually performed. A screenshot of Historef is shown in Fig. 4. This example shows a situation in which a developer added some comments and applied a rename refactoring to a specific source code. The Changes view on the right side of the figure shows the information of all changes including their content and executed time. Historef has a feature of undo/redo history refactorings because it permits developers' trial-and-error in history restructuring.

Historef has a feature of change grouping for supporting complex configuration of history refactorings. In Historef, developers explicitly assert their focused group by specifying *editing modes* for each edit operation during their code editing process. Developers modify the source code using a


Resource - dpd_sample1/src/Main.java - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Selective Undo command
History refactoring commands
Resource
Main.java
Changes
Groups
Groups view
Changes view
Mode 3/3: adding comm
Writable
Smart Insert
21 : 27

```
public static void main(String[] args) {
    try {
        Directory rootDir = new Directory("root
        Directory dir = new Directory("dir1");
        File file2 = new File("file1", 100);
        file2.addObserver(new SizeObserver());
        rootDir.add(file2);
        rootDir.add(dir);
        dir.add(new File("file2", 200));
        dir.add(new File("file3", 300));
        Link link = new Link(file2);
        dir.add(link);

        int i = 1;
```

Group	Time	Change
1	07/14 14:22:55.640	#1: add(702, " ")
1	07/14 14:22:59.921	#2: add(730, "int i = 1; ")
1	07/14 14:23:08.062	#3: remove(888, " ")
1	07/14 14:23:08.375	#4: remove(888, "")
1	07/14 14:23:09.906	#5: remove(742, " ")
− 2	07/14 14:23:19.312	Changeset [3 chunks]
	07/14 14:23:19.312	#9: replace(1085, "link1", "link")
	07/14 14:23:19.312	#10: replace(695, "link1", "link")
	07/14 14:23:19.312	#11: replace(649, "link1", "link")
1	07/14 14:23:30.359	#12: add(741, " ")
3	07/14 14:23:54.093	#13: add(755, "// showing results")

ID	Caption
☐ 1	fixing [bug:4]
☐ 2	Rename Local Variable
☑ 3	adding comments

Figure 4. Screenshot of Historef.

special editor having multiple editing modes [4]. All groups are listed in the Groups view. Newly added changes performed on the editor automatically belong to the active group selected in the Groups view. Developers can fix the group of past changes at any time. To avoid the cost of a group definition, Historef can define a new group automatically for large changes of several types. For example, Group 2 (a code refactoring) was created automatically using the name of the edit operation corresponding to the automated rename refactoring in Eclipse.

Applications. Using the history refactorings implemented in Historef, two applications are available. The first one is the support of the task level commit. By reordering edits and merging them according to their group using Reorder, users can commit entangled edits to underlying SCM repository to achieve the task level commit. The second one is the *selective undo* of changes. Basically, the normal undo feature revokes only the most recently performed edit operation. In contrast, the selective undo feature [8] enables users to undo any change without limiting the timing when they are executed. First, the focused changes are moved to the recent using Swap. Next, they are merged into a single change using Merge. Finally, the recent change is executed inversely to remove the effect of the focused changes from the current source code. Historef has a special command in the Changes view for automating these operations simultaneously. Developers can undo all past changes using this command if the entire related history refactorings succeed.

IV. Related Work

Because of the space limitation, we compare only some important techniques with ours. Although some approaches have already accomplished rewriting of a software development history (such as Git, Mercurial, or Darcs), the proposed approach is novel because it defines the rewriting of history based on the existing concept of refactoring and focuses not on revision histories but on edit histories, and implements a supporting tool that seamlessly collaborates with IDE. Change models and development environments based on them for understanding and reuse have been proposed [2], [9]. However, tools provided by them are based mainly on structure editor-based changes. These changes are unsuitable for activities of practical developers, who write source code using trial-and-error on a source code editor.

V. Conclusion and Future Work

As described in this paper, we proposed a technique for refactoring edit histories and its supporting tool Historef. Although no usability evaluation of Historef has been performed yet, we showed that the proposed history refactorings are feasible and showed their benefits with two useful applications.

An attractive benefit for defining the rewriting of the edit history using existing concept of refactoring is to reuse relevant concepts of refactoring. To build large refactorings by the composition of primitive refactorings is an example in this paper. Another example is to define *bad smells* in the edit history. An automatic edit checker can be built by detecting smells before committing if we can define impropriety measures of an edit history as history smells. For example, changes corresponding with a specific group can be considered Shotgun Surgery in an edit history. To define smells of edit history is left as future work.

In addition, improving the usability of Historef, e.g., automated support for the detection of groups, is also left as future work.

Acknowledgements. This work was partly sponsored by KAKENHI Grants (23700030, 24500050, and 24700034).

References

[1] L. Hattori, M. Lungu, and M. Lanza, "Replaying past changes in multi-developer projects," in *Proc. IWPSE-EVOL*, 2010, pp. 13–22.

[2] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 93–109, 2007.

[3] S. Berczuk and B. Appleton, *Software Configuration Management Patterns*. Addison-Wesley, 2002.

[4] S. Hayashi and M. Saeki, "Recording finer-grained software evolution with IDE: An annotation-based approach," in *Proc. IWPSE-EVOL*, 2010, pp. 8–12.

[5] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proc. ICSE*, 2011, pp. 351–360.

[6] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE TSE*, vol. 38, no. 1, pp. 5–18, 2012.

[7] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," in *Proc. MSR*, 2008, pp. 31–34.

[8] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects," *ACM Trans. Computer-Human Interaction*, vol. 1, no. 3, pp. 269–294, 1994.

[9] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt, "Change-oriented software engineering," in *Proc. ICDL*, 2007, pp. 3–24.