Hierarchical Categorization of Edit Operations for Separately Committing Large Refactoring Results

Jumpei Matsuda, Shinpei Hayashi, and Motoshi Saeki
Department of Computer Science
Tokyo Institute of Technology
Tokyo 152–8552, Japan
{jmatsu,hayashi,saeki}@se.cs.titech.ac.jp

ABSTRACT

In software configuration management using a version control system, developers have to follow the commit policy of the project. However, preparing changes according to the policy are sometimes cumbersome and time-consuming, in particular when applying large refactoring consisting of multiple primitive refactoring instances. In this paper, we propose a technique for re-organizing changes by recording editing operations of source code. Editing operations including refactoring operations are hierarchically managed based on their types provided by an integrated development environment. Using the obtained hierarchy, developers can easily configure the granularity of changes and obtain the resulting changes based on the configured granularity. We confirmed the feasibility of the technique by applying it to the recorded changes in a large refactoring process.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*

General Terms

Algorithms, Theory

Keywords

Refactoring, edit history, tangled changes

1. INTRODUCTION

In software development using a version control system (VCS), changes following the *commit policy* of the project are worthful for developers. Developers often understand the code differences performed by other developers, and such understandings help them to understand the meaning of changes and to reuse them [12]. A commit policy defines the discipline of commits by developers. e.g., the upper bound of the size of a commit or the number of meaningful changes included in a commit. For instance, the policy of the Task Level Commit [5] is well known and useful. For improving the understandability and/or reusability of changes, Task Level Commit prohibits committing changes of mixed intentional edits. It requires developers to commit changes, each of which includes only one intention.

However, in actual software development, keeping changes following the commit policy is not a trivial task. Many pieces of research [3, 4, 6, 15–17, 19] reported that *tangled changes*, i.e., changes including multiple intentional edits, often occur when developers prepare their commits. In order to follow the used commit policy, developers are required to pay attention to editing their source code carefully in appropriate manner and order and to committing the performed edits on appropriate moments. Although some VCSs have the features to merge commits or split a commit into multiple ones, these features are not intended to prepare appropriate commits from the edits performed by developers, and they require a redundant understanding of the performed changes, which is time-consuming.

In particular, a complicated editing process often happens when applying a large refactoring. A large refactoring consists of multiple primitive refactoring instances. Developers are needed to apply multiple refactoring operations during a large refactoring process if the whole refactoring is not automated by refactoring tools. In addition, developers often apply *floss refactoring* [18]; refactorings are applied together with non-refactoring changes. In such a case, the resulting sequence of changes are tangled; it includes refactoring operations of different types and non-refactoring edits. Packing these changes as a single commit should be avoided due to the low understandability of the resulting change content.

In this paper, we propose a technique for hierarchically grouping edit operations of source code in order to retrieve sets of operations by specifying the granularity of commits according to the used commit policy. In our technique, each branch of the built hierarchy corresponds to a commit of certain granularity. By collaborating with existing integrated development environment (IDE), some special operations including automated refactorings and non-essential changes [16] can be easily recorded and identified. Our technique automatically builds a hierarchy of such operations and utilizes them for constructing commits of multiple levels of granularity. Developers can obtain the changes according to the specified branch nodes by reordering the performed edit operations. Since our reordering procedure carefully

IWPSE'15, August 30, 2015, Bergamo, Italy
ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804363

constructs the changes based on the dependency between edit operations, the consistency of the resulting changes is guaranteed when the procedure succeeds.

We have implemented our technique as a plug-in of Eclipse IDE [1]. We regard the refactoring operations provided by Eclipse as the intentional meanings of changes and their similarity of types as the intentional relationship between changes. By applying our technique to an edit history including both automated and manual edit operations, we show the effectiveness of our approach.

The rest of this paper is organized as follows. In the next section, we clarify our background and motivation using an example. Section 3 describes our approach, and Section 4 illustrates the supporting tool implementing our approach. We discuss our case study in Section 5 to show the effectiveness of our approach. Section 6 introduces related work, and we conclude this paper and state our future work in Section 7.

2. BACKGROUND

In software configuration management (SCM), developers commit their changes. A commit has a message that expresses its meaning or purpose in addition to the change content representing how the change modifies the source files. Developers have to understand the commits of other developers from the message and/or the content of the commits.

One factor why tangled changes occur is the load to prepare commits according to the meaningful intentions separately. The meaning and the size of a commit is determined when a developer performs a commit based on the edits from the most recent revision to the present. One of the ways to construct commits for each intention is to edit source code in the specific order for making changes of each task at once. However, this is an annoying and time-consuming task because at least two editing tasks may relate each other. Although existing VCSs enable developers to perform a commit by selecting some of the edited chunks from the whole edits, it is hard for developers to select only appropriate chunks having the same intentional meaning after performing all of the edits.

Therefore, it is useful for developers to reconfigure their changes after performing them when the resulting changes do not follow the compliant commit policy. We call this process *dividing commits*. An illustrative example of the dividing commits is shown in Figure 1. In this figure, edit operations of three different tasks are performed: change of the naming convention, the aggregation of duplicated code, and a fix according to a change request. The left side of the figure shows the fragments of source code before and after applying the edit operations; we can have the left bottom fragment of code when applying the edit operations shown in the right side to the code fragment at the left top of the figure. First, by applying Rename Method refactorings, the names of methods were changed that the new names have a specific prefix in order to follow the changed naming convention. Next, two `if` statements, which are the duplicates, are aggregated into one using Extract Method refactoring. Finally, a behavioral fix according to the given change request was done. For brevity, its related edits are omitted as "$\cdots$" as the last edit operation in the figure. These changes happen when we apply floss refactorings [18]; when implementing a fix for a change request, such changes have frequently happened in practice [19].

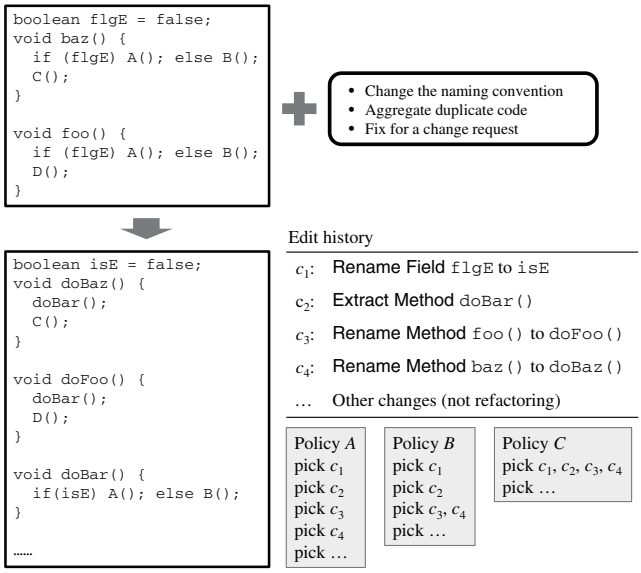


Figure 1: Illustrative example of dividing commits.

How we commit these edits differs according to the commit policy that the target software project follows. For example, consider a commit policy that refactoring operations of different types should be committed separately as different intention of changes in order to describe the commit log as using the type of refactoring as the intention of the change (named Policy B). In contrast, we can also consider another policy that allows us to commit coarser-grained changes for only separating the behavior-preserving changes such as refactorings with the other non-behavior-preserving ones (named Policy C). Moreover, a finer-grained policy compared to Policy B that prohibits to commit multiple refactoring instances at once for improving the understandability of the differences of changes (named Policy A). In Policy A, in addition to the separation based on intentions, developers have to commit each of refactoring instance included in a sequence of intentional changes separately; the policy prohibits to commit so-called *impure* refactoring [9]. This kind of policy leads to increase the number of commits but the ease of the understanding and/or validation of each commit.

Actually, some commit policies define how refactoring-related changes should be divided like the above policies in practice. For example, the commit policy of an open source project named OpenStack [2] suggests to avoid committing mixing functional code changes with whitespace changes: "*The whitespace changes will obscure the important functional changes, making it harder for a reviewer to correctly determine whether the change is correct*". It also suggests to separate refactoring and non-refactoring changes: "*It is highly desirable that any refactoring is done in commits which are separate from those implementing the new feature. This helps reviewers and test suites validate that the refactoring has no unintentional functional changes*". The separation of refactoring and non-refactoring changes improves the understandability of changes when reviewing the resulting commits [8, 13, 23].

Consider that we have a sequence of edits as shown in the table in Figure 1. The lines shown in the right bottom of

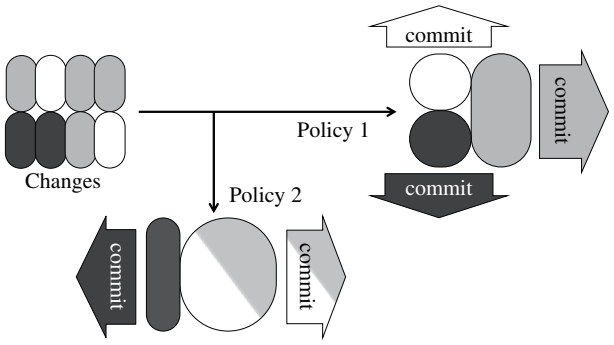


Figure 2: Basic idea.

the figure indicate how the commits in each policy should include which edit operations. For example, the edit operations $\{c_1, c_3, c_4\}$ (a fix for following the new naming convention by Rename Method refactoring), $\{c_2\}$ (the aggregation of code by Extract Method refactoring), and $\{\cdots\}$ (the other changes) should be committed separately in Policy B. This separation does not follow the other policies, i.e., Policies A and C. In Policy C, we have to separate the edit operations like $\{c_1, c_2, c_3, c_4\}$ and $\{\cdots\}$, and in Policy A we have to do like $\{c_1\}$, $\{c_2\}$, $\{c_3\}$, $\{c_4\}$, and $\{\cdots\}$. Committing all the edits shown in the figure does not follow any of these policies. It means that we have to prepare the changes of different granularity based on the compliant commit policy.

Below, we call the granularity of the changes for following the policy the *granularity of a policy*. For example, Policy B treats the applied refactorings as non-behavior-preserving changes and does not split them into multiple refactoring commits. In contrast, Policy A distinguishes the applied refactorings that have the same type from the refactorings of the other types. In other words, Policy A can treat more concrete intentions of changes. In addition, Policy A is finer-grained than Policy B in terms of the granularity of the following changes; the changes following Policy A are finer-grained than those following Policy B.

3. PROPOSED TECHNIQUE

3.1 Our Approach

We propose a technique for reorganizing changes according to different commit policies. Our technique records changes on a structure and enables users to organize changes in different kinds of granularities in order to make the resulting commits follow the granularity of the used commit policy. This makes developers easy to adjust the granularity with keeping the consistency of changes. Since a refactoring-related editing process is the target of this paper, we use the type of refactoring operations for representing the meaning of recorded edit operations.

Figure 2 shows the basic idea of our technique. We record edit operations of source code as finest-grained changes using IDE and reorganize them into meaningful changes according to the used commit policy. If the development project follows a commit policy preferring finer-grained changes, the recorded edit operations are reorganized to finer-grained changes. In contrast, if the development project follows a commit policy preferring coarser-grained changes, the recorded edit operations are merged and reorganized to coarse-grained changes. On one hand in Figure 2, eight finest-grained changes are recorded, and they are categorized into three groups, specified by the colors. In Policy 1, these changes are merged according to their groups, and three commits are organized. On the other hand in Policy 2, since white-colored and gray-colored groups are regarded to the same editing intention, all the changes of these two groups are merged, and coarser two commits are organized. The point is that commits are organized after all the edit operations are performed, which mitigates the cost of developers for paying attention to the manner and order of their code editing.

The overview of our technique is illustrated in Figure 3. We regard an editing operation such as lexical addition or deletion to source code by a developer as a fine-grained change to be recorded. The recorded changes are reordered according to their intentions. For the separation of changes by intentions, we introduce a hierarchical grouping method, which handles a group as a node and a meaningful parent-child relationship as an edge between nodes. The node closer to the root node has the meaning of more abstract than those far from the root node. We convert a commit policy to a set of nodes in the hierarchy representing how we construct commits. After selecting the nodes based on the commit policy, we extract the ordering relation between nodes and reorder the changes belonging to the nodes so that we realize the commits appropriate for the given policy. Finally, the obtained changes are committed to underlying SCM repository such as Git.

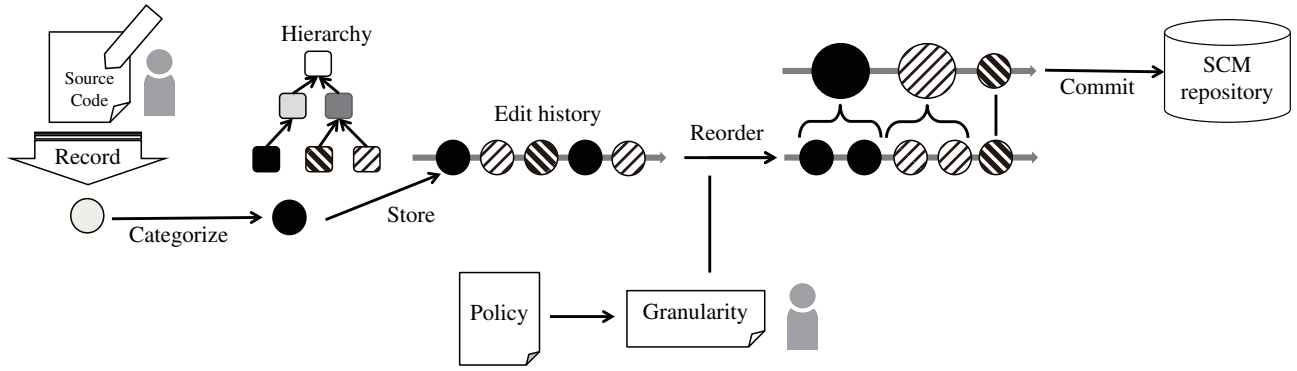


Figure 3: Overview of the technique.

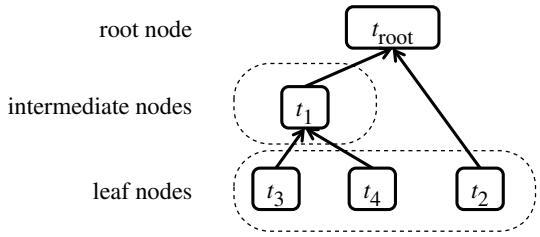


Figure 4: Example of the tree structure.

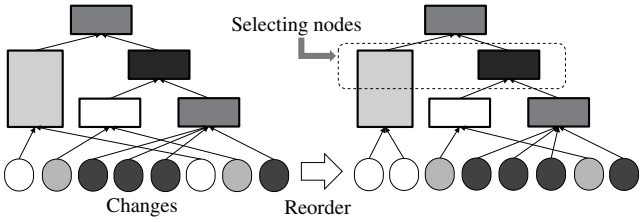


Figure 5: Reorganizing changes.

3.2 Definitions

In our technique, the edit operations of source code are recorded by OperationRecorder [21] and managed by Historef [11]. The definitions among them are based on Historef methodology. We extend the original definition in order for dealing with a hierarchal structure instead of sequential one that the original methodology intended to use.

We call a series of addition, removal, or replace of a specific source code file *chunk*. A chunk $h := (t, f, o, r, a)$ is a tuple representing a modification of the characters on a file of source code, where:

- t is the time when the edit operation was performed,
- f is the file to be edited,
- o is an integer representing the starting offset of the edit,
- r is the removed string from the edited file, and
- a is the added string to the edited file.

Each element of an edit history $H := c_1 c_2 \dots c_N$ is a change. A change is a pair of a sequence of chunks $h_1 h_2 \dots h_n$ and a group $g \in G$ to which the change belongs: $c = (h_1 h_2 \dots h_n, g)$. Each chunk and the size of a change can be respectively referred as $c[i] := h_i$ and $|c| := n$. Some of the dependencies between changes exists. For example, consider an edit history $H = c_{\text{add}} c_{\text{remove}}$ where c_{add} is a change of the addition of string "abc", and c_{remove} is another change of the removal of the string "abc" added by c_{add}. Here, it is impossible to swap these changes because the latter change deletes the text that is introduced by the former one, and we call this situation that the latter change depends on the former change.

We introduce a hierarchical grouping that extends the existing one. The hierarchical grouping consists of a tree structure. We define the node of the hierarchy $t \in T$ corresponding to the group g as $t = (id, caption, t_{parent})$ and redefine a change c as $c := (h_1 h_2 \cdots h_n, t)$, where:

- id is the identifier of the node,
- $caption$ is the caption string representing the name of the node, and
- t_{parent} is the parent of the node.

The root node does not have a parent node t_{parent} for avoiding cycling.

As mentioned above, the set of nodes in the hierarchical groups T forms a tree structure. Figure 4 shows an example of such tree structure of the hierarchical groups. This tree structure consists of five nodes: t_{root}, t_1, t_2, t_3, and t_4. The node $t_{root} = (root, caption, t_{root})$ is the *root node* of the tree structure. $T_{leaves} \subset T$ denotes the *leaf nodes* who do not have any child nodes. A node $t \in T$ is called *intermediate* node if it is neither root nor leaf node: $t \notin T_{leaves} \cup \{t_{root}\}$. For example in Figure 4, t_{root} is the root node, t_1 is the intermediate node, and t_2, t_3, and t_4 are the leaf nodes.

All the changes belong to leaf nodes of the tree structure. When a change c belongs to a leaf node t, the relation between the change and the node can be referred by $\mathrm{t}(c) := t$. Although no changes directly belong to intermediate nodes, they indirectly belong to them based on the parent-child relationship between nodes. Therefore, intermediate nodes can be regarded as a kind of abstract categories. All of the changes C_t that belong to a node t and its descendant nodes can be referred by $\text{changes}(t) := C_t$. For example, an instance of Rename Field refactoring to the field f using a feature of an IDE belongs to the node $t_f^{\text{Rename Field}}$, and the actual change representing the replacement of each occurrence of the corresponding identifier can be retrieved by $\text{changes}(t_f^{\text{Rename Field}})$.

Our interest is how to divide the set of all the changes in the leaf nodes into smaller sets of changes. We denote a *division* of T_{leaves} by $T_{sep} \subseteq 2^{T_{leaves}}$, with satisfying the following condition:

$$\forall t_1 \neq t_2 \subseteq T_{sep} \bullet t_1 \cap t_2 = \emptyset,$$
$$T_{leaves} = \bigcup_{t \in T_{sep}} t.$$

This condition guarantees that T_{sep} covers every element of T_{leaves} and its two optional elements are disjoint.

Since all the changes belong to only the leaf nodes, the order on the leaf nodes is important when reordering changes. A binary relation $\prec \subseteq T_{leaves} \times T_{leaves}$ expresses the total order for reordering the changes on the proposed tree structure. We can prepare this order relation $\prec$ based on the given separation T_{sep}. For example, if two nodes t_1 and t_2 are classified to the same category, i.e., $\{\dots, t_1, t_2, \dots\} \in T_{sep}$, occurring of a node t in another category in T_{leaves} in between these nodes in the order is prohibited: $t_1 \prec t \prec t_2$ or $t_2 \prec t \prec t_1$ should not hold.

3.3 Reorganizing Changes

We reorganize the changes by reordering the edit operations based on the given hierarchy in order to construct the commits following the suitable granularity defined by the used commit policy. Figure 5 shows the overview of reorganizing changes. This procedure consists of the following two steps. First, we select nodes of the given hierarchy according to the suitable granularity. Next, we reorder the edit operations in the order based on the selected nodes. These two steps realize to organize the commits following the policy.

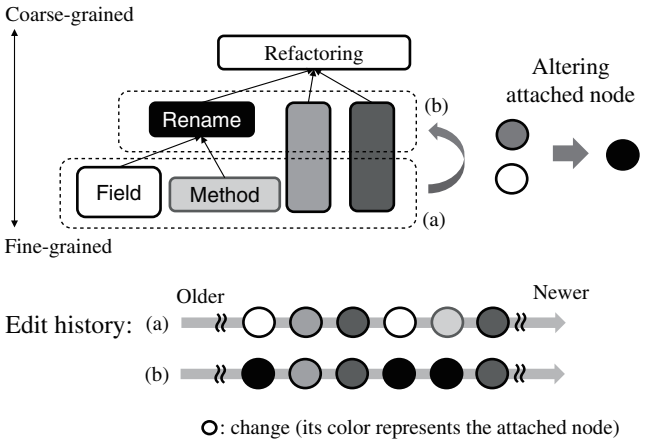


Figure 6: Selecting nodes.

Name: ReorderH$(H, H', \preceq)$
Input:

$H \equiv c_1 \cdots c_N$: edit history
$H' \equiv c_i \cdots c_j$: target subsequence of H
$\prec \subseteq T_{leaves} \times T_{leaves}$: order relation of leaf nodes

Steps:

1. Sort H' by the bubble sorting based on $\prec$ in order to satisfy the condition $\mathrm{t}(c_i) = \mathrm{t}(c_k) \Rightarrow \mathrm{t}(c_i) = \mathrm{t}(c_j)$ for all $i < j < k$. Swap is used for swapping two changes in the sort.
2. If it is necessary to merge changes, it does. When merging the changes into one, it executes Merge$(H, c_i \cdots c_j)$ for all $i \cdots j$ s.t. $\mathrm{t}(c_i) = \cdots = \mathrm{t}(c_j)$.

Name: Swap(H, c_i, c_j)
Input:

$H \equiv c_1 \cdots c_N$: edit history
c_i, c_j: Adjacent target changes in H

Steps:

1. $c'_i \leftarrow c_i$, $c'_j \leftarrow c_j$.
2. **for** $k = |c'_i|$ **downto** 1 **do**;
   for $l = 1$ **to** $|c'_j|$ **do**;
   Commute $c'_i[k]$ and $c'_j[l]$ [12].
3. $H \leftarrow c_1 \cdots c_{i-1} c'_j c'_i c_{j+1} \cdots c_N$.

Name: Merge$(H, c_i c_{i+1} \cdots c_j)$
Input:

$H \equiv c_1 \cdots c_N$: edit history
$c_i c_{i+1} \cdots c_j$: target subsequence of H

Steps:

1. $c' \leftarrow (c_i[1] \cdots c_i[|c_i|] c_{i+1}[1] \cdots c_{i+1}[|c_{i+1}|] \cdots c_j[1] \cdots c_j[|c_j|], \mathrm{t}(c_i))$.
2. $H \leftarrow c_1 \cdots c_{i-1} c' c_{j+1} \cdots c_N$.

Figure 7: Procedures ReorderH, Swap, and Merge.

3.3.1 Selecting Nodes

Selecting the granularity of commits for reorganizing is required when reorganizing commits according to the used commit policy. In our technique, this step is equivalent to selecting nodes. Figure 6 illustrates the change of target granularity by selecting nodes. A circle and a horizontal arrow in the figure respectively represent a change and an edit history. In the edit history, newer changes are located in the right-side. The figure includes an excerpt history consisting of six changes: $c_1, \ldots, c_6$. In Figure 6(a), the user had selected four nodes, t_2, t_3, t_4, and t_5, i.e., four groups of edit operations, and we classify the changes by the selected four types. Then, the user discontinued to treat the rename of methods and fields as different changes and regarded them as an atomic change of abstract and coarser-grained renaming. The nodes of Rename Method and Rename Filed belong to the node of abstract Rename. Changes are belonging to the node of Rename Method or Rename Filed. As shown in Figure 6(b), the user selected coarser-grained nodes, t_1, t_2, and t_3, that regard the renaming as a single intention. By changing the nodes, all of the changes were classified into three groups. Actually, three changes that initially classified into two groups, c_1, c_4, and c_5, were regarded as having the same intention and classified into a single group.

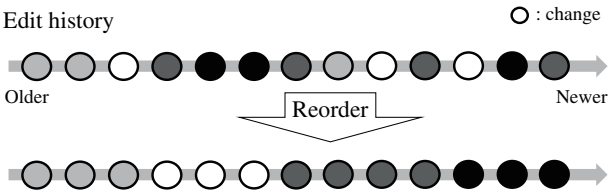


Figure 8: Example of reordering using ReorderH.

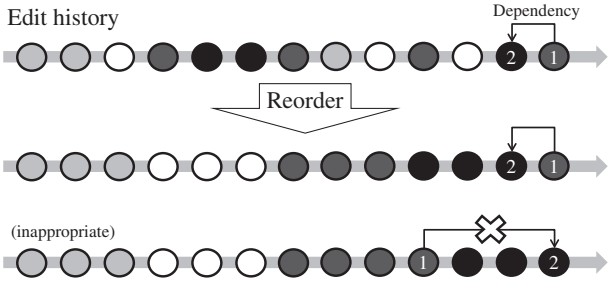


Figure 9: Dependency constraint.

3.3.2 Reordering Edit Operations

We defined another history refactoring operation that reorders the changes on an edit history categorized by the proposed hierarchy. The defined operation ReorderH is an extension of Reorder, the original reordering operation in Historef [11]. We extended Reorder in order to treat hierarchical groups as input and to use the order relation between the leaf nodes of the hierarchy.

The procedure of ReorderH is shown in Figure 7. ReorderH uses two history refactorings defined in Historef: Swap and Merge. Swap swaps two given changes. Merge merges given changes into a single change. By using ReorderH, the given subsequence of the edit history H' is reordered and the changes that belong to the same element of T' are merged into a single change. The resulting merged changes follow the order relation $\prec$. Therefore, the granularity of the resulting changes are defined according to the given order $\prec$. Figure 8 shows the ideal reordering when applying ReorderH to the

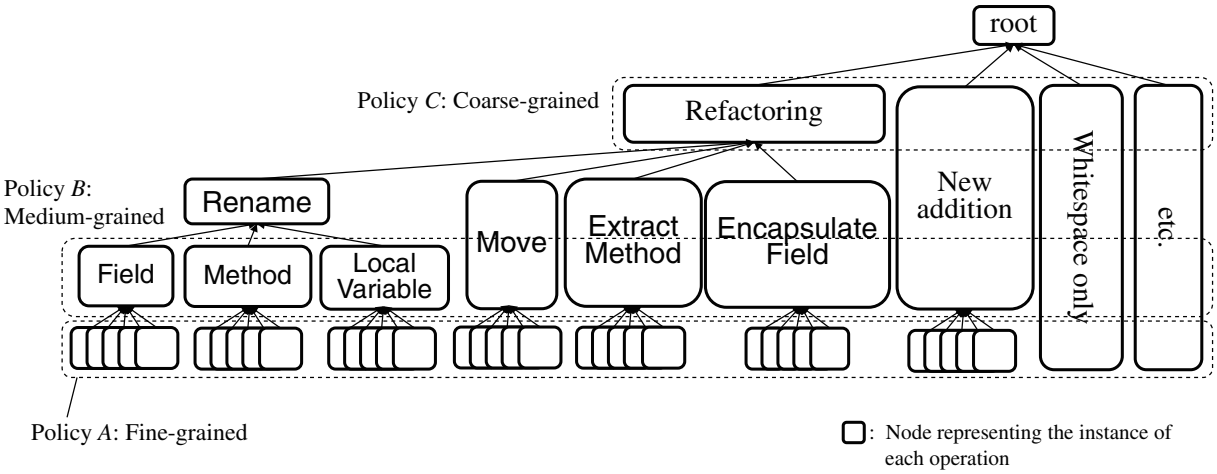


Figure 10: Hierarchical groups.

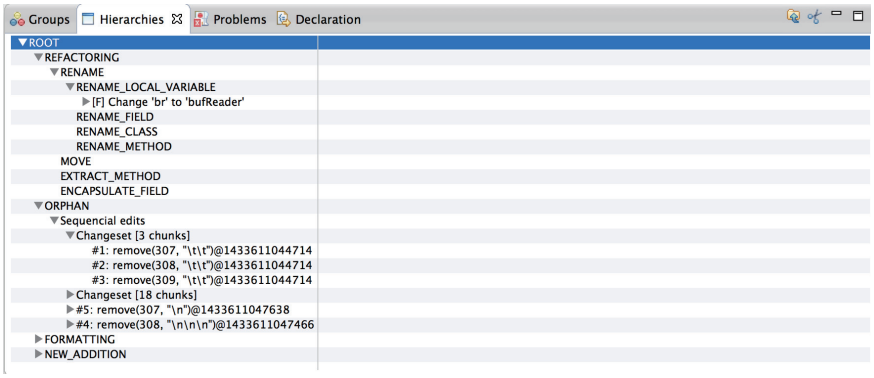


Figure 11: Screenshot of the tool.

edit history H. The arrows of the vertical direction indicate the execution of ReorderH. The changes of the same color will be located to close and adjacent.

Note that there are some dependency relationships between changes. The reordering using ReorderH has to keep the relative order between two changes among a dependency. For example, consider the dependencies shown in Figure 9. As the resulting history of ReorderH, the bottom history is inappropriate because it violates the dependency constraint between the changes 1 and 2, and the middle one is the actual result.

4. IMPLEMENTATION

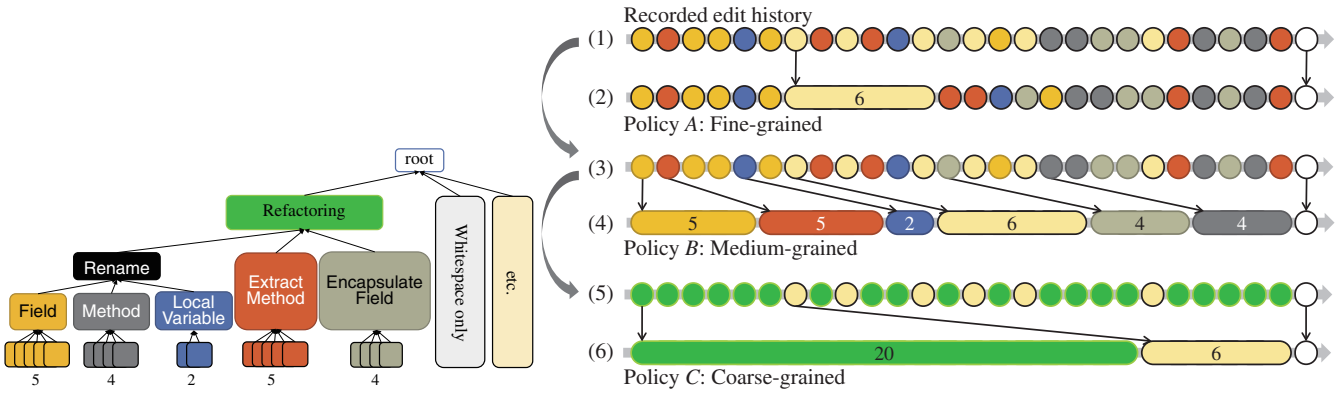
We have implemented a tool for providing an automated support of our approach. The edit operations used in our technique are collected using OperationRecorder [21], which captures the edit operations executed on Eclipse IDE [1]. Our tool is an extension of existing history refactoring tool named Historef [10, 11]. When OperationRecorder sends an edit operation to Historef [11], an associated appropriate leaf node is selected based on the information of the recorded edit operation. For example, when a refactoring operation is conducted, the caption of the node can be extracted from the target and the type of the refactoring operation.

Our tool is able to capture six popular refactoring operations that are provided by Eclipse IDE and that fit on the hierarchy structure: Rename Field, Rename Method, Rename Local Variable, Move, Extract Method, and Encapsulate Field. Also, we prepared some additional nodes. The first one is for representing the automated whitespace insertion and deletion, which are used for formatting source code. The second one is for representing the addition of new classes and methods. The last one is specialized for classifying the other changes that could not belong to any other nodes.

Figure 10 shows the resulting tree structures of the generated nodes. Its root node has four child nodes. The first child is the *refactoring*-node that represents the entire refactoring operations. The second child is the *whitespace*-node that represents the automated editing of white spaces. The third child is the *new-addition*-node that represents the creation of new classes and methods. The last child is the *etc.*-node that represents the other operations. The whitespace- and etc.-nodes are regarded as leaves.

The refactoring-node consists of several nodes that represent the type of refactoring operations in parent-child relationships. We also provided the rename-node that represents the entire refactoring about renaming since there are several types of renaming refactorings. Rename-node has three children: ones for Rename Field, Rename Method, and Rename Local Variable.

A screenshot of the implemented tool is shown in Figure 11. This screenshot shows a visualized hierarchy when



(a) Built hierarchy. (b) Reorganization of edit operations.

Figure 12: Application results.

Table 1: Numbers of Commits

Policy	# expected commits	# actual commits
A	22	22
B	12	12
C	3	3

applying a Rename Local Variable refactoring and the other non-refactoring changes. The conducted edit operations are automatically shown in the tree view during the editing process of the developer. This view also allows users to correct the tree structure. Users can fix the category of changes and to add new nodes when finding some mistakes about the categorization. The dividing commits feature can be invoked by selecting nodes from the given hierarchy.

5. APPLICATION EXAMPLE

5.1 Target

We have applied our technique for an edit history of a Java file included in the project developing a Twitter client for Android OS. The file includes 3903 lines of code, 29 of fields, and 105 of methods. One of the authors applied large refactoring and manual edits to the code because it had several maintainability problems such as a duplicated code fragment and the violation of the coding conventions on variable names. From the editing process during 40 min, we obtained an edit history consisting of 37 changes and 796 editing chunks. The applied refactorings include five of Rename Field, four of Rename Method, two of Rename Local Variable, four of Encapsulate Field, five of Extract Method, a formatting, and six of the other non-essential changes.

The source file were edited using the implemented tool explained in Section 4 so that we obtained the corresponding hierarchy. Figure 12(a) shows the obtained hierarchy. This hierarchy consists of 22 leaf nodes.

5.2 Results

We applied our technique to this history using the prepared three commit policies shown in Section 2 and obtained the divided commits for each policy. The dashed squares in Figure 10 indicate which nodes are captured by each policy. Policies *A*, *B*, and *C* are respectively regarded as fine-, medium-, and coarse-grained ones.

Figure 12(b) shows how the obtained edit operations were reordered and merged for each policy. In the figure, the color of changes represents the node to which it belongs and corresponds to the color of the node of the hierarchy in Figure 12(a). The number in a change is the total number of the leaf nodes to which the change belongs. The black arrows of the vertical direction between changes specify the non-trivial associations before and after the change of the node selection. The edit histories (1), (3), and (5) are respectively the ones according to the fine-, medium-, and coarse-grained commit policies before applying change reordering, whereas (2), (4), and (6) are the results after applying change reordering.

As a result, the change reordering for each policy succeeded without any conflicts, and we could obtain the appropriate changes for each policy. Table 1 shows the numbers of expected and actual commits for each policy. The number of the obtained commits was the same as that of the expected commits for each policy.

As shown in Figure 12, we can see that different commit policies produce different sets of commits. Policy *A* (fine-grained) produced the changes similar to the original ones. In Policy *B* (medium-grained), some changes were reordered, and the number of the produced changes then increased. Since Policy *C* (coarse-grained) is the coarsest-grained, the number of the reordered changes is less than that of Policy *B* (medium-grained).

5.3 Discussion

For the used commit policies of three types, it was possible to divide commits at different granularities. We confirmed that the change contents of each commit were consistent with the meaning indicated by the belonging node that constitutes the hierarchy. We also confirmed that the size of the obtained commits was suitable in accordance with each selected commit policy. These facts indicate that the proposed technique can divide commits if we use a refactoring-aware commit policy as mentioned above. Regarding refactoring operations, commit log messages obtained using Policy *A* were approximately equivalent to the actual editing process, which applies five different types of refactoring operations irregularly. This means that we can obtain the same sequence of commits if committing every after applying a refactoring operation. However, such flow forces developers to be inter-

rupted during their large refactoring process and is really time-consuming.

Policy B summarizes several renaming changes as a single commit. Therefore, Rename Field refactoring operations, which were distributed at second and thirteenth commits in Policy A, are located in close. This fact indicates that we are required to apply refactoring operations in the different order in order to manually obtain the same sequence of commits.

6. RELATED WORK

Study of tangled changes and techniques of untangling them are proposed. For example, Herzig *et al.* reported that 33.8% of commits are tangled in average [15]. Nguyen *et al.* also reported that 11–39% of fixing commits include other changes [20]. There are several types of untangling approaches including metric-based [6, 15], change template-based [17], and dependency-based [4,20]. Also, a proposal of an interactive environment for manipulating changes allows developers to reorganizing commits [3]. However, most of them do not focus on the preference of granularity of changes and hierarchical relationship of refactoring operations.

Techniques for detecting the instance of refactoring operations from changes are proposed [22,23], and detected results can be used for improving the understandability of source code differences [8, 13]. In our technique, we obtain the information of refactoring operations from IDE only, and the results of manual refactoring are missed. By analyzing the contents of changes, we can put the resulting edit operations of manual refactorings on our hierarchy.

Some refactoring-aware configuration management techniques [7, 14] capture and replay refactoring operations in order to make them portable. When reordering the changes in our technique, swapping of two changes fails if a textual dependency between them is found. Our approach is complementary to their approaches; using such replaying mechanism to our dependency checking might improves the applicability of our history reordering.

7. CONCLUSION

In this paper, we proposed a technique for reorganizing changes according to the refactoring-related commit policy that the target development project follows. In our technique, we record the edit operations of source code by developers and categorize them in a hierarchy based on the types of the operations. By selecting the nodes of the hierarchy as a reordering criterion and reordering the operations, we obtain the set of changes appropriate for committing according to the used commit policy. We have implemented a supporting tool as a plug-in of Eclipse, which enables us to build change hierarchy and reorganize changes based on the hierarchy. An application of our technique consisting of different kinds of refactorings was conducted to confirm whether we could obtain different sets of commits according to different commit policies. As a result, we could obtain different sets of commits following different granularities of policies.

The future work can be summarized as follows.

Evaluation. In this paper, we confirmed only whether appropriate changes are organized or not. We are planning to conduct a human study for checking the usability and the actual effort using our tool.

Richer hierarchy. The current types of node in our hierarchy are limited. We will define other types of nodes for categorizing edit operations in various way.

8. ACKNOWLEDGMENTS

This work was partly supported by JSPS Grants-in-Aid for Scientific Research (Nos. 15H02685 and 15K15970).

9. REFERENCES

[1] Eclipse. `http://www.eclipse.org/`.

[2] Git commit good practice - OpenStack wiki. `https://wiki.openstack.org/wiki/GitCommitMessages`.

[3] T. Barik, K. Lubick, and E. Murphy-Hill. Commit bubbles. In *Proc. 37th International Conference on Software Engineering* (*ICSE 2015*), pages 631–634, 2015.

[4] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. 37th International Conference on Software Engineering* (*ICSE 2015*), pages 134–144, 2015.

[5] S. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.

[6] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *Proc. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (*SANER 2015*), pages 341–350, 2015.

[7] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. 29th International Conference on Software Engineering* (*ICSE 2007*), pages 427–436, 2007.

[8] X. Ge, S. Sarkar, and E. Murphy-Hill. Towards refactoring-aware code review. In *Proc. 7th International Workshop on Cooperative and Human Aspects of Software Engineering* (*CHASE 2014*), pages 99–102, 2014.

[9] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proc. 13th International Workshop on Program Comprehension* (*ICPC 2005*), pages 205–214, 2005.

[10] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama. Historef: A tool for edit history refactoring. In *Proc. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (*SANER 2015*), pages 469–473, 2015.

[11] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki. Refactoring edit history of source code. In *Proceedings of the 28th IEEE International Conference on Software Maintenance* (*ICSM 2012*), pages 617–620, 2012.

[12] S. Hayashi and M. Saeki. Recording finer-grained software evolution with IDE: An annotation-based approach. In *Proc. Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution* (*IWPSE-EVOL 2010*), pages 8–12, 2010.

[13] S. Hayashi, S. Thangthumachit, and M. Saeki. Rediffs: Refactoring-aware difference viewer for java. In

Proceedings of the 20th Working Conference on Reverse Engineering (*WCRE 2013*), pages 487–488, 2013.

[14] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proc. 27th International Conference on Software Engineering* (*ICSE 2005*), pages 274–283, 2005.

[15] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. 10th International Workshop on Mining Software Repositories* (*MSR 2013*), pages 121–130, 2013.

[16] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proc. 33rd International Conference on Software Engineering* (*ICSE 2011*), pages 351–360, 2011.

[17] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *Proc. the 22nd International Conference on Program Comprehension* (*ICPC 2014*), pages 262–265, 2014.

[18] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5), 2008.

[19] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[20] H. A. Nguyen, A. T. Nguyen, and T. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Proc. 24th IEEE International Symposium on Software Reliability Engineering* (*ISSRE 2013*), pages 138–147, 2013.

[21] T. Omori and K. Maruyama. A change-aware development environment by recording editing operations of source code. In *Proc. 5th Working Conference on Mining Software Repositories* (*MSR 2008*), pages 31–34, 2008.

[22] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. 26th International Conference on Software Maintenance* (*ICSM 2010*), 2010.

[23] S. Thangthumachit, S. Hayashi, and M. Saeki. Understanding source code differences by separating refactoring effects. In *Proceedings of the 18th Asia Pacific Software Engineering Conference* (*APSEC 2011*), pages 339–347, 2011.