Historef: A Tool for Edit History Refactoring

Shinpei Hayashi[π], Daiki Hoshino[π], Jumpei Matsuda[π], Motoshi Saeki[π], Takayuki Omori[R], and Katsuhisa Maruyama[R]
[π] Department of Computer Science, Tokyo Institute of Technology
Ookayama 2–12–1–W8–83, Meguro-ku, Tokyo 152–8552, Japan
Email: {hayashi,dhoshino,jmatsu,saeki}@se.cs.titech.ac.jp
[R] Department of Computer Science, Ritsumeikan University
Nojihigashi 1–1–1, Kusatsu-shi, Shiga 525–8577, Japan
Email: {maru@,takayuki@fse.}cs.ritsumei.ac.jp

***Abstract*—This paper presents Historef, a tool for automating edit history refactoring on Eclipse IDE for Java programs. The aim of our history refactorings is to improve the understandability and/or usability of the history without changing its whole effect. Historef enables us to apply history refactorings to the recorded edit history in the middle of the source code editing process by a developer. By using our integrated tool, developers can commit the refactored edits into underlying SCM repository after applying edit history refactorings so that they are easy to manage their changes based on the performed edits.**

***Keywords*—software evolution; refactoring; tangled changes**

I. Introduction

Well-managed edit histories help developers significantly. 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 [1]. In accordance with the task level commit, efforts of each task can be adopted or reverted flexibly, and we can manage the changes intuitively with understandable granularity. In addition, a number of open source software projects have adopted a patch-based flow. In order to realize the submitted patches to be accepted, developers are forced to make the patches easy to understand 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, it is not a trivial task for keep edit histories well-managed. Developers often make various kinds of changes in one term. For instance, a developer who notices a design issue in fixing some bugs would merge edits for refactoring and bug-fixes [2], [3], [4]. In such situation, changes corresponding with multiple tasks are often performed at once in a working copy. As a result, the difference based on the obtained edits does not adapt task level commit and needs to be fixed.

To bridge this gap, we have proposed a technique of *edit history refactoring* [5]. An edit history refactoring, inspired by source code refactoring, is a rewriting of edit history of source code for improving usability and/or understandability without changing its effect. This paper introduces its automated tool named Historef, which enables us to manage edit operations by recording them and applying history refactoring to them. Although there are some existing techniques for untangling changes [6], [7], this tool is novel because they do not provide an interactive tool for bridging this gap between commits and edits.

The remainder of this paper is organized as follows. In the next section, we briefly introduce the technique of edit history refactoring [5], which is the basis of Historef. Section III introduces how we use Historef. The architecture and the current limitation of Historef are respectively explained in Sects. IV and V. Finally, the paper is concluded by Sect. VI.

II. Edit History Refactoring

A history refactoring [5] is defined as a restructuring of edit history for improving the usability and/or understandability of the history without changing its whole effect. This means that 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. History refactorings are performed before sharing their changes with other developers. In order to avoid repetitions or mistakes of understanding, we should not restructure a history that has been already shared with other developers.

We have defined primitive refactorings of several types with their pre- and post-conditions. Swap swaps the execution order of target changes. Merge composes multiple changes into a single change. Split splits a change into multiple changes. Also, by composing of multiple primitive refactorings, large refactorings can be defined. For example, Reorder reorders the sequence of changes from a tangled edit history according to their belonging groups, which is based on multiple applications of Swap.

III. Usage of Historef

A. Overview

We have implemented Historef—a supporting tool for automating history refactorings in Java development[1]. The process of code refactoring tool use includes not only the execution of refactorings but also its configuration and undo [8]. Also for history refactoring tools, these supports are desired because they permit developers' trial-and-error in history restructuring. We defined the following requirements for Historef:

- applying history refactorings to an edit history,
- supporting the configuration of history refactorings, and
- undo/redo of history refactorings.

[1] Available from http://www.se.cs.titech.ac.jp/historef/

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

SANER 2015, Montréal, Canada

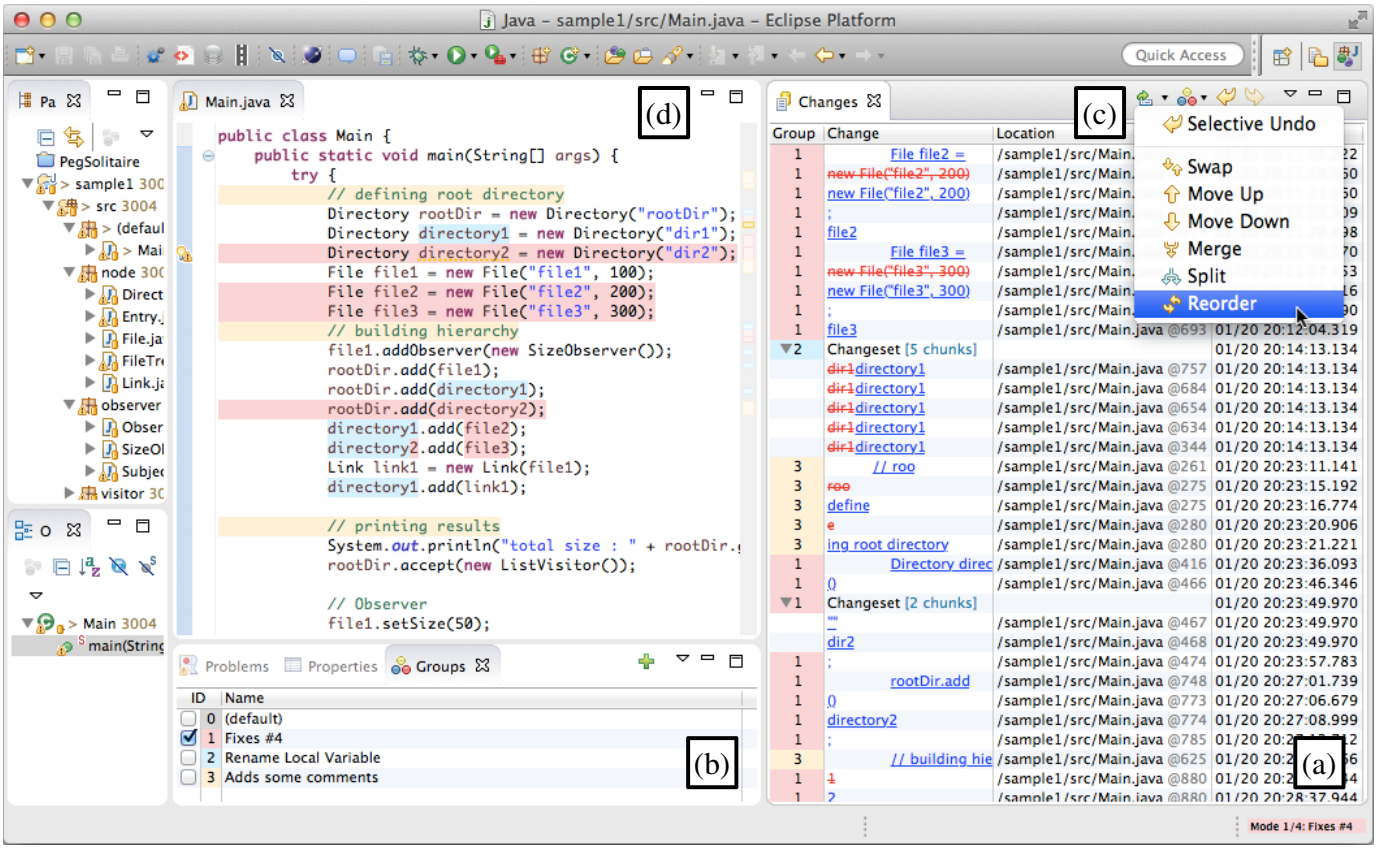


Fig. 1. Screenshot of HISTOREF. (a) Changes view; it immediately shows the performed changes by the owner developer. (b) Groups view; it manages the groups of changes. (c) Commands of history refactorings, selective undo, and the undo/redo of them. (d) Source code editor highlighting the recorded edits shown in Changes view with colors.

Additionally, for improving the usability of history refactorings, the following features are also useful:

- the support of automated grouping of edits and
- collaboration with underlying SCM repository.

A screenshot of HISTOREF is shown in Fig. 1. This figure shows a situation that 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 managed change history including all the performed edits with their content and executed time. This history can be refactored using the provided history refactoring commands.

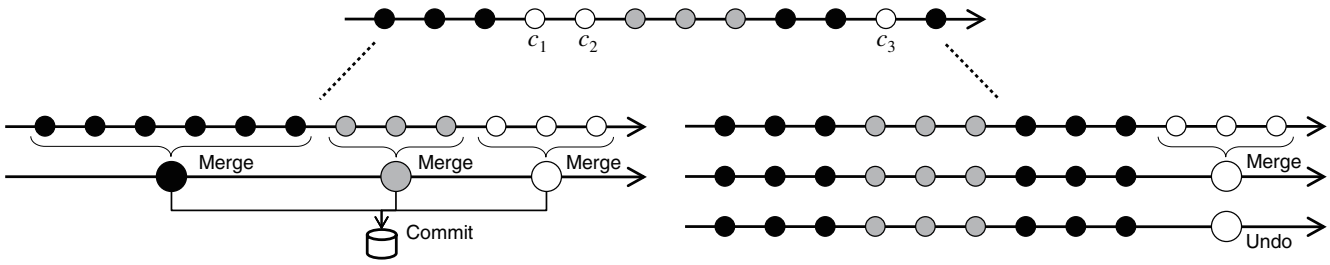
Using the history refactorings implemented in HISTOREF, the following two applications are available.

Task level commit. By reordering edits and merging them according to their group, developers can commit untangled edits to underlying SCM repository in order to achieve the task level commit (Fig. 2(a)). It is achieved by committing deltas of merged changes after applying Reorder. Figure 3 shows the Changes view after applying Reorder refactoring to the history shown in Fig. 1. Changes are reordered according to their groups and users can commit the changes belonging to each group separately. Note that the resulting order of groups in Fig. 3 is not straightforward because HISTOREF detected the textual dependency between changes and found an appropriate order of changes.

Selective undo. Similarly to Azurite [9], an automated *selective undo* mechanism is provided. The normal undo feature revokes only the *recent* edit operation. In contrast, the selective undo feature enables users to undo any changes without limiting their executed timing. An overview of the selective undo feature is shown in Fig. 2(b). This feature is also based on the reordering of edit history. First, the focused changes (the target of undo; c_1, c_2, and c_3 in this example) are moved to the recent using Swap. Next, they are merged into one change using Merge. Finally, the recent change is inversely executed to remove the effect of the focused changes from the current source code. HISTOREF has a special command for automating these operations at once. Developers can undo all of the past changes using the selective undo command if all the related history refactorings succeed.

B. Manual Manipulation of History

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 special editor having multiple editing modes [10]. A developer is free to switch the current editing mode from 0 (by default) to whatever mode using shortcut keystrokes. The result of the switch is notified immediately to the developer, who can modify the source code with the awareness of the current



(a) Task level commit. (b) Selective undo.

Fig. 2. Applications of history refactorings. Arrows, circles, and their colors respectively express edit histories, changes on them, and their belonging groups.

Group	Change	Location	Time
▼2	Changeset [5 chunks]		01/20 20:14:13.134
	~~dir1~~directory1	/sample1/src/Main.java @691	01/20 20:14:13.134
	~~dir1~~directory1	/sample1/src/Main.java @601	01/20 20:14:13.134
	~~dir1~~directory1	/sample1/src/Main.java @554	01/20 20:14:13.134
	~~dir1~~directory1	/sample1/src/Main.java @534	01/20 20:14:13.134
	~~dir1~~directory1	/sample1/src/Main.java @344	01/20 20:14:13.134
3	// roo	/sample1/src/Main.java @261	01/20 20:23:11.141
3	~~roo~~	/sample1/src/Main.java @275	01/20 20:23:15.192
3	define	/sample1/src/Main.java @275	01/20 20:23:16.774
3	~~e~~	/sample1/src/Main.java @280	01/20 20:23:20.906
3	ing root directory	/sample1/src/Main.java @280	01/20 20:23:21.221
3	// building hie	/sample1/src/Main.java @466	01/20 20:28:01.966
3	// printing res	/sample1/src/Main.java @823	01/20 20:28:49.842
1	File file2 =	/sample1/src/Main.java @466	01/20 20:11:05.222
1	~~new File("file2", 200)~~	/sample1/src/Main.java @680	01/20 20:11:18.560
1	new File("file2", 200)	/sample1/src/Main.java @493	01/20 20:11:23.060
1	;	/sample1/src/Main.java @515	01/20 20:11:25.309
1	file2	/sample1/src/Main.java @703	01/20 20:11:28.898
1	File file3 =	/sample1/src/Main.java @516	01/20 20:11:46.170
1	~~new File("file3", 300)~~	/sample1/src/Main.java @766	01/20 20:11:57.453
1	new File("file3", 300)	/sample1/src/Main.java @543	01/20 20:12:00.416
1	;	/sample1/src/Main.java @565	01/20 20:12:02.090
1	file3	/sample1/src/Main.java @789	01/20 20:12:04.319
1	Directory direc	/sample1/src/Main.java @416	01/20 20:23:36.093
1	()	/sample1/src/Main.java @466	01/20 20:23:46.346
▼1	Changeset [2 chunks]		01/20 20:23:49.970
	""	/sample1/src/Main.java @467	01/20 20:23:49.970
	dir2	/sample1/src/Main.java @468	01/20 20:23:49.970
1	;	/sample1/src/Main.java @474	01/20 20:23:57.783
1	rootDir.add	/sample1/src/Main.java @783	01/20 20:27:01.739
1	()	/sample1/src/Main.java @808	01/20 20:27:06.679
1	directory2	/sample1/src/Main.java @809	01/20 20:27:08.999
1	;	/sample1/src/Main.java @820	01/20 20:27:12.712
1	1	/sample1/src/Main.java @880	01/20 20:28:37.944

Fig. 3. Edit history after applying Reorder to the one shown in Fig. 1.

editing mode. Each mode is associated with a specific group. For example, the developer fixes a bug using mode 1, while applying a code refactoring in the act of the bug-fix by (newly created) mode 2. The former change belongs to the group 1, while the latter one does to the group 2. In the example shown in Fig. 1, three groups including 1: bug-fix, 2: code refactorings, and 3: addition of comments are defined in addition to the default group, and several changes belong to these groups. All the groups are listed up in the Groups view. Newly added changes performed on the code editor automatically belong to the active group selected in the Groups view. Developers can fix the belonging group of past changes at any time.

Developers can add a name to groups. The names can be used for an initial input of commit log description when they commit the resulting changes into a SCM repository.

In order to avoid the cost of group definition, HISTOREF can automatically define a new group for several types of large changes. For example, the changes in group 2 (Rename Local Variable refactoring) are executed using the code refactoring feature in Eclipse. This group was automatically created using the name of the edit operation corresponding with the rename refactoring.

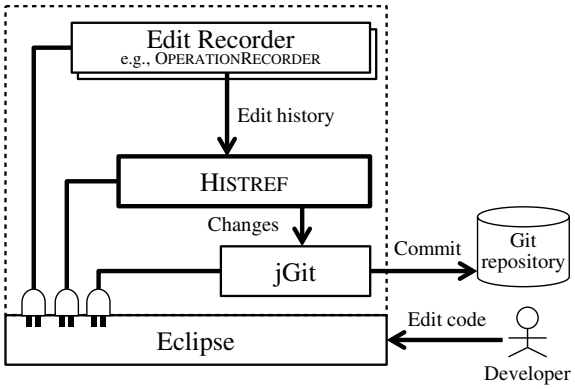


Fig. 4. Architecture of HISTOREF.

C. Automated Grouping

Instead of manual grouping of edits, several criteria of automated grouping are provided. By applying the *automated grouping* feature, developers can obtain the edits classified based on the selected criterion. We provide criteria based on temporal and/or syntactical information: *Time-Based*, *File-Based*, *Class-Based*, *Method-Based*, and *Comment-Based*. For example, two edits performed within the predefined threshold time will be classified as the same group by *Time-Based* criterion, whereas edits modifying the same method will be classified as the same group by *Method-Based* criterion (See Fig. 6).

For syntactical classification, it is necessary to find the precise position in the abstract syntax tree (AST) of each edit operation. We used ASTParser of Eclipse JDT as syntactic analysis. Note that we must fix the offset of each edit operation because its offset can change due to subsequent edit operations. Similarity to the approaches by Omori *et al.* [11], [12] and Negara *et al.* [13], our tool automatically fixes the offset of edits by analyzing the addition or removal ranges of subsequent edits.

IV. ARCHITECTURE

HISTOREF is designed as a plug-in for Eclipse and is collaborating with edit recorder plug-ins such as OPERATIONRECORDER [11] or FLUORITE [14], which collect the edits that the owner developer of Eclipse actually performed. The architecture of HISTOREF is shown in Fig. 4. The connected

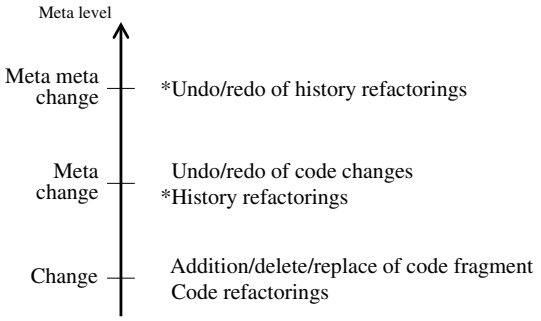


Fig. 5. Changes on HISTOREF. The commands marked by "*" are the ones unsupported by Eclipse.

edit recorder extracts edit operations of Java source code, and the extracted history is transferred to HISTOREF core module. HISTOREF constructs changes from the recorded edit operations. Users can commit the constructed changes to the underlying Git repository using jGit module.

HISTOREF manages on a different level of history from the one managed by the Eclipse. Figure 5 shows how HISTOREF deals with code changes. In the original editor of Eclipse, edit operations are stored in its undo buffer and are the target of undo/redo. In contrast, HISTOREF manages *meta changes* which modify an edit history for achieving undo/redo of history refactorings.

After obtaining an edit operation from an edit recorder, HISTOREF constructs a meta change having expressing the addition of the edit operation to the edit history, and adds this meta change to the managing meta change history. History refactorings, which modify an edit history, are also defined as meta changes; they are also added to the meta change history when executed. HISTOREF prepares undo/redo operations in the meta-change level. Similarly to the original Eclipse code editor in the change level, HISTOREF enables developers modify their edit history by trial-and-errors. The undo/redo mechanism in the change level is achieved by undoing/redoing a meta change that adds a change. For this reason, HISTOREF does not have to provide the undo/redo command of the change-level as a specific meta change. Note that HISTOREF provides a special meta change that removes the recent change from the edit history for achieving the selective undo mechanism.

V. LIMITATIONS

The current implementation of HISTOREF has several limitations:

- If the associated edit recorder misses some changes, and the resulting history includes inconsistency, the history refactorings may cause inappropriate commits or incorrect undo results.
- Our change model does not include the operations of a project, such as the addition or removal of files in a project. These operations are out of the scope of our history refactorings.
- HISTOREF uses its own undo history without using the one managed by Eclipse. The features of history refactorings and selective undo are performed for this history. For this reason, it is hard to collaborate with other features provided by other plug-ins that use the original undo history of Eclipse.

VI. CONCLUSION

In this paper, we proposed HISTOREF, a tool for refactoring edit histories. Although the usability evaluation of HISTOREF have not been performed yet, we showed that the proposed history refactorings are feasible, and they benefits two useful applications.

We will continue to improve the quality of HISTOREF for mitigating the limitation shown in the last section. In particular, we plan to extend our model of edit history for managing some complex structures of changes, such as branches or hierarchical relationships, so that our tool will have an ability to utilize changes in more realistic context.

ACKNOWLEDGMENTS

This work was partly sponsored by the Grant-in-Aid for Scientific Research (C) (24500050) and for Young Scientists (B) (23700030, 26730042) from the Japan Society for the Promotion of Science (JSPS).

REFERENCES

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

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

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

[4] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. MSR*, 2013, pp. 121–130.

[5] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring edit history of source code," in *Proc. ICSM*, 2012, pp. 617–620.

[6] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *Proc. ISSRE*, 2013, pp. 138–147.

[7] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proc. ICPC*, 2014, pp. 262–265.

[8] E. Murphy-Hill, "A model of refactoring tool use," in *Proc. WRT*, 2009.

[9] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of fine-grained code change history," in *Proc. VL/HCC*, 2013, pp. 119–126.

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

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

[12] K. Maruyama, E. Kitsu, T. Omori, and S. Hayashi, "Slicing and replaying code change history," in *Proc. ASE*, 2012, pp. 246–249.

[13] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *Proc. ECOOP*, 2012, pp. 79–103.

[14] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proc. PLATEAU*, 2011, pp. 25–30.

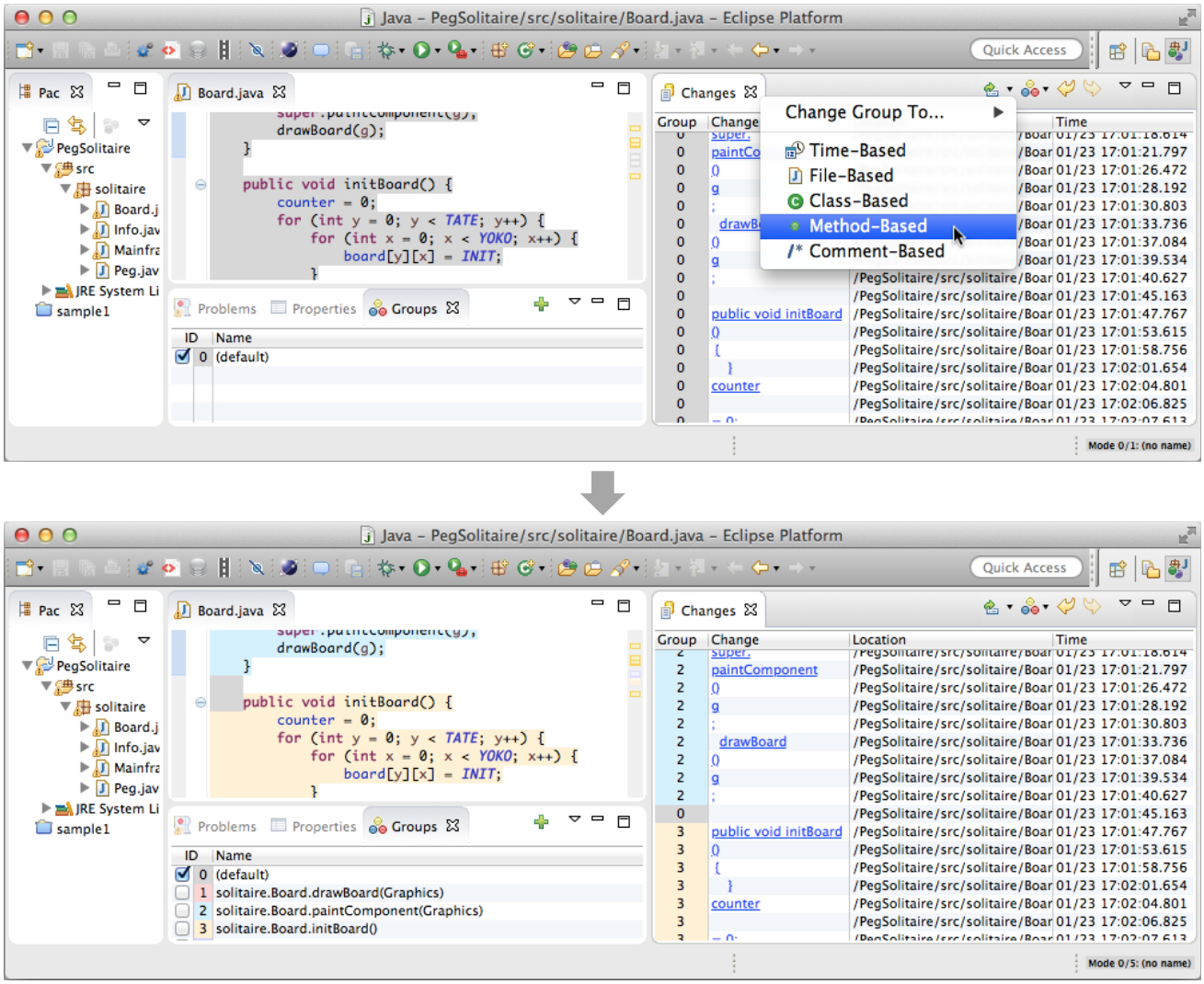


Fig. 6. Example of automated grouping. In this example, a developer added several methods including `paintComponent` and `initBoard` to the class `Board`, and applied automated grouping by using *Method-Based* criterion. Applying the grouping, the edits related to `paintComponent` and `initBoard` are respectively classified as the second and third groups in Changes view.