

An Accurate and Convenient Undo Mechanism for Refactorings

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University
1-1-1 Noji-higashi Kusatsu Shiga 525-8577, Japan
maru@cs.ritsumeai.ac.jp

Abstract

Refactoring makes existing source code more understandable and reusable without changing observable behavior. Therefore, applying refactorings to existing source code and reversing the effects of past refactorings are inseparable from tool support to software development and maintenance. This paper proposes a powerful undo mechanism that uses a chain of past refactorings for each source file and determines which refactoring is undoable by monitoring the last refactoring in every chain. The proposed undo mechanism can make an undesired refactoring accurately undone although the refactoring has affected multiple source files. Moreover, it permits a programmer to compatibly execute operations of refactoring undo and editorial undo/redo without the limitation of manual edit of files. A running implementation of the mechanism has been integrated into our developed refactoring browser.

1. Introduction

Refactoring[9, 3] is a behavior-preservation transformation and facilitates programmers making existing source code more understandable and reusable. It is thus essential and useful practice in developing and maintaining object-oriented software, and modern integrated development environments (IDEs) tend to include a refactoring tool or browser. In Java programming language, several IDEs including Eclipse [2] or IntelliJ IDEA [5] support refactoring¹. With automated (or semi-automated) tools, the programmers are fully or partially relieved from tedious checks and accidental errors of source code. Moreover, refactoring helps maintainers to understand unfamiliar code written by others since it allows the maintainers to actually change the code to reflect their ideas how the code works, and lets them view different code having the same behavior as before.

However, a refactoring tool cannot in general guarantee every automated refactoring to truly preserve the behavior

of source code since a precise definition of behavior is rarely provided or might be inefficient in practice [8]. In addition, actual improvement of existing source code cannot be performed by only a limited kind of automated refactorings. Therefore, manual modifications are often required before or after applying the refactorings. In this case, several automated and manual transformations are mixed in the process of improvement. The effect of the improvement much depends on the order of applied refactorings.

From the point of view of no guarantee of behavior-preservation and no determinate plan of applying refactorings, a refactoring tool should provide a mechanism to make erroneous or undesired refactoring undone [10]. It is easy to implement such undo mechanism if the changes of an applied refactoring are always enclosed in one source file. Therefore, conventional undo mechanisms assume that changes are simply managed in units of source files. However, many refactorings tend to rewrite multiple source files at the same time, nevertheless none of existing refactoring tools provide an undo mechanism that can correctly manage coincident and consecutive changes of multiple source files.

This paper presents a powerful undo mechanism to reverse the changes dispersed by past refactorings with the consistency between restored and existing source files. The undo mechanism assigns sequence numbers to all past refactorings to identify them and records the identification numbers into a chain of refactoring undo for each source file. Undoing a refactoring never leads to unexpected cancellation of changes derived from other refactorings by determining which undo operations are feasible based on the identification numbers at the end of respective chains. Moreover, the undo mechanism can deal with manual changes made by programmers, and permits them to compatibly execute operations of both refactoring undo and editorial undo/redo. A running implementation of the mechanism has been integrated into our developed refactoring browser. The significant contribution of this paper is to present a simple but nontrivial undo mechanism that can reverse more changes by refactorings and to demonstrate its concrete design and implementation.

¹Many up-to-date tools and tools supporting other languages can be seen in <http://www.refactoring.com>.

By providing this accurate and convenient undo mechanism, the programmers (and maintainers) can comfortably apply automated refactorings to improve the design of deteriorated code resulting from repeated modifications. In addition, the proposed undo mechanism helps maintainers to obtain experimental variants of the original code to verify their understanding of the code. If they want to see other experimental code, they would undo the applied refactorings and then apply different refactorings to the restored code.

The remainder of the paper is organized as follows: Section 2 describes a general refactoring process and explains a problem arising from refactoring undo. Section 3 proposes a new undo mechanism using chains of past refactorings, and Section 4 explains its design and implementation. Section 5 describes studies related to refactoring tools with undo facilities. Finally, Section 6 concludes with a brief summary and describes remaining issues.

2. Refactoring Undo

This section describes a refactoring process including undo activity, and explains a typical problem arising from operations of undo refactorings. Moreover, it presents conventional ideas that try to address the problem and their remaining issues.

2.1. Refactoring Process

To preserve the observable behavior of existing code in refactoring, a large change of the code is performed by applying a series of small, safe transformations. Transformations in refactoring can be mainly shown in the studies by Opdyke and Fowler. Opdyke proposed 26 low-level refactorings and three high-level ones [9]. Fowler proposed a catalog of 72 refactorings, consisting of names, motivations, mechanics, and examples [3].

The refactoring process consists of the following six steps [8]:

1. Detect bad smells in the software and identify where the code should be refactored;
2. Determine which refactoring should be applied to the identified code fragments;
3. Check preconditions that should be satisfied prior to the determined refactoring;
4. Apply the refactoring by executing a number of transformations. If the smells partially remain, go back to the step 2;
5. Assess the effects of the refactoring(s) on quality characteristics of the software.
6. Maintain the consistency between the refactored code and other software artifacts.

In real software development, tool support is considered crucial although the code is possible to refactor manually. Thus, almost all IDEs provide the refactoring functionality that automates the tasks to check preconditions and to change the code in the steps 3 and 4.

Refactoring undo assumes that all or parts of refactorings are applied by an automated refactoring tool. The undo activity is usually executed in the step 4 or 5. In the end of the step 4, a programmer might reverse the applied refactoring if he/she feels that it was improper for removing the detected smells. In the end of the step 5, he/she wants to restore code before applying the refactoring(s) if the desired effect was not obtained or the quality was decreased.

To easily understand examples that will be mentioned later, several refactorings are illustrated here. A `MOVEMETHOD` refactoring moves a method in the class on which it is defined to another class that frequently uses the method. This refactoring in general creates a new method with the same or a similar body in the target class, and turns the old method into a simple delegation or removes it. Similarly, a `MOVEFIELD` refactoring moves a field in the class on which it is defined to another class that frequently uses the field. This refactoring creates a new field in the target class, and changes all its references. A `RENAMEMETHOD` or `RENAMEFIELD` refactoring changes the original name of a method or field into an appropriate one, respectively. These refactorings will be applied in the case that the name of a method or field does not reveal its purpose. These four refactorings are all automated by popular IDEs.

2.2. Motivating Examples

It is significant for all refactoring tools to provide the functionality of undoing undesired past refactorings, nevertheless few tools implement refactoring undo and every undo mechanism integrated into those tools has been still unsatisfactory. An ordinary implementation of a refactoring undo mechanism is based on the restricted linear undo [1], which is simply extended so that it can reverse the whole change dispersed in different source files. The restricted linear undo is often provided by popular source code editors and its mechanism performs a cancellation of an immediately preceding editorial action (insertion, deletion, replacement, cut, copy, or paste of a text) by storing past actions in a linear list of each file.

Figure 1 shows several versions of each of three source files X, Y, and Z, and the results of undoing some refactorings. The X_n , Y_n , and Z_n ($n = 1, 2, \dots$) represent the contents of the files. In Figure 1(a), a programmer applied a `MOVEMETHOD` refactoring to the files X and Y (e.g., moving a method $m()$ from a class of X to Y), and then a `RENAMEFIELD` refactoring to the file Z (e.g., renaming a method $n()$ of a class of Z). Once he/she executes an undo operation to X in this situation, the last refactoring of X (`MOVEMETHOD`) will be reversed, and then the contents

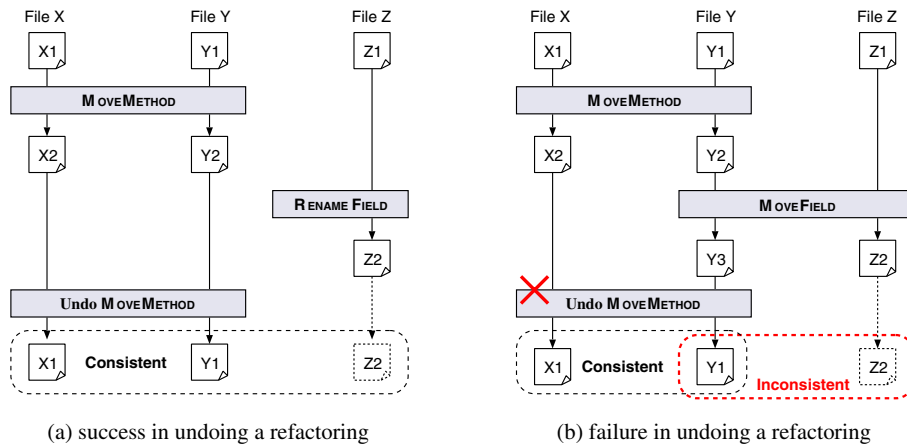


Figure 1. Undo a refactoring affecting multiple files.

of X and Y will be restored to X1 and Y1, respectively. The reason why Y1 will be recovered is that Y was related to X in the undone refactoring. This undo operation is feasible. (The undo operation to Y or Z is also feasible.)

However, in case that different refactorings applied to source files X, Y, and Z are tangled as shown in Figure 1(b), the simple undo mechanism will probably violate the consistency of the contents of these files. If the contents of X and Y are simply restored to X1 and Y1, the inconsistent combination of Y1 and Z2 will happen since the contents Y1 and Z2 are not permitted to coincidentally exist. For example, if a method $m()$ was moved from a class of X to Y in a MOVEMETHOD refactoring and then a field f was moved from a class of Y to Z in a MOVEFIELD refactoring, both Y1 and Z2 after undoing would contain the same field f . If f was moved from a class of Z to Y in the MOVEFIELD refactoring, neither Y1 nor Z2 contains f after undoing. Consequently, no programmer accepts this undo operation.

2.3. Conventional Ideas

Roberts showed three ways to implement the undo facilities for a refactoring [11]. The first way is the use of an inverse refactoring. The idea is simple but all inverse operations are difficult to prepare as refactorings. Therefore, this mechanism is rarely used to implement the undo facilities.

The second way is to make a checkpoint after applying each refactoring. The effects of past refactorings are recorded together in the same chain. Undoing a refactoring is performed by canceling the last effect in the chain. This mechanism can be simply implemented. In fact, most refactoring tools including Eclipse provide this undo mechanism. However, the mechanism has a disadvantage in user-friendliness because undoing a refactoring in the middle of the chain requires undoing some refactorings that were performed later in the chain. For example, Eclipse permits undoing only the last of past refactorings. In Figure 1(a), a

programmer cannot undo the MOVEMETHOD refactoring prior to the undo of the RENAMEFIELD refactoring. This limitation is inconvenient to many programmers.

To alleviate this kind of inconvenience, the third way utilizes dependencies of refactorings done in the past. This mechanism simply undoes all refactorings that were performed later than the refactoring to be undone, undoes the target refactoring, and then reapplies the later refactorings. If any of the reapplied refactorings is illegal, the mechanism stops undoing the target refactoring. This mechanism has been actually integrated into the Smalltalk Refactoring Browser [11]. Although it is more likely to be able to undo an arbitrary refactoring, it cannot be considered suitable for many refactoring tools. For future undo, this undo mechanism must capture information for locating a code fragment (e.g., a method name in a Rename Method refactoring) which is specified by a programmer. A class and its members (a method and a field) are relatively easy to locate since they can be identified by using their names. However, it is much difficult to locate a statement without its identifier or a local variable redundantly appearing in one method, if a refactoring changes the location (offset) of the statement or the local variable within the refactored code. Therefore, it is almost impossible to reapply a refactoring involving such change (e.g., a Split Temporary Variable refactoring in Fowler's catalog [3]). Moreover, general source code editors make source files have respective liner lists (buffers) for undoing editorial changes. It is unclear and might be unsolved that how the third way deals with the two types of undo buffers for refactoring and editing, and how it preserves the consistency between results of refactoring undo and editorial undo/redo.

3. Undo Using Refactoring Chains

To avoid both of the inconvenience due to the limitation in applying an undo operation and the troublesome (or im-

possible) reapplication of refactorings, the new undo mechanism ingeniously uses the chains of past refactorings. This section defines undoable refactorings and proposes a new undo mechanism using them.

3.1. Undoable Refactorings

The proposed mechanism assigns identification numbers to all refactorings applied in the past and chains several refactorings related to each source file among the applied refactorings in order of their applications. When a programmer applies a refactoring to a source file, the undo mechanism generates a new identification number (unique sequence number), adds the number to the refactoring undo chain of each of the refactored source files, and records the set (collection) of the refactored source files. The recorded file set can be obtained by using the identification number of a specified refactoring as a retrieval key. The undo mechanism determines which refactorings are undoable (or whether undoing a specified refactoring is valid) and activates an undo command (a menu item on a source code editor) for the undoable refactoring, by monitoring the last refactoring in the chains of source files.

The definition of an undoable refactoring is as follows:

Undoable refactoring: A set of all the source files affected by a refactoring r is denoted by S_r . The refactoring r is **undoable** if all the chains for the source files in the set S_r contain the same identification number indicating the refactoring r at their ends (i.e., the last identification numbers of all the chains are the same).

A refactoring that does not satisfy the above definition is called a non-undoable refactoring. When the programmer wants to execute an operation of undoing the last refactoring r applied to a source file, the undo mechanism checks if r satisfies the above condition. If r is an undoable refactoring, the mechanism cancels the last change of each source file in the file set S_r and restores the contents of all the source files to those before the application of r . Then, it removes the last identification number from the chains for the source files. If r is a non-undoable refactoring, the mechanism does not activate an undo command so that the programmer cannot execute the command.

Figure 2 shows an undoable refactoring and a non-undoable refactoring, which are shown in Figure 1(b). The symbols (#1 and #2) attached to past refactorings mean their identification numbers. The programmer can make the MOVEFIELD:#2 refactoring undone when editing the source file Y or Z because both the identification numbers at the end of the chains for the two files are #2; on the contrary, he/she cannot undo the MOVEMETHOD:#1 refactoring because the source files X and Y have different identification numbers #1 and #2 at the end of their chains. If MOVEFIELD:#2 is undone in advance, the undo of MOVEMETHOD:#1 will be permitted.

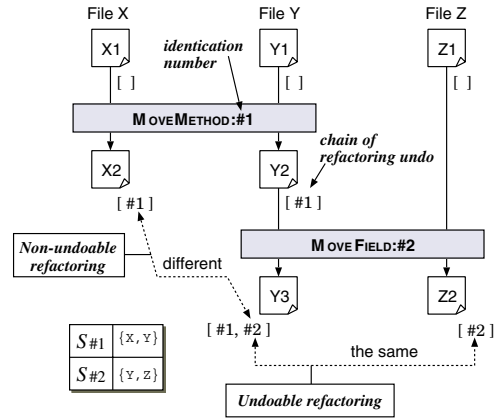


Figure 2. Undoable and non-undoable refactorings.

The order of the refactorings recorded in the chain of each source file is consistent with the order of all refactorings applied to the whole source code in the past. Therefore, the undo mechanism guarantees the order of undone refactorings for each source file not to change.

3.2. Relationships between Different Files

The new undo mechanism accepts only undo operations for undoable refactorings mentioned Section 3.1. At a glance, this mechanism seems to guarantee behavior preservation of existing source code. However, this is true as far as the interference arising from refactorings is explicitly reflected in the refactored source files. The implicit interference exists in practical object-oriented programs and it sometimes causes violation of the consistency between restored and existing source files.

In Java programming language, there are the following possible relationships between different source files:

1. Package. Classes defined in different files can be included in the same package by using the **package** clause.
2. Inheritance. A class defined in one file extends or implements a class defined in another file by using the **extends** or **implements** clauses.
3. Method invocation. The code existing in one file calls a method of a class defined in another file by using the dot operator. The method invocation is classified as static, non-static without polymorphism, or non-static with polymorphism.
4. Field access. The code existing in one file refers to a field of a class defined in another file by using the dot operator. The field access is classified as static, non-static without polymorphism, or non-static with polymorphism.

dition stems from the Java grammar that more than one class with the same name is forbidden to exist in one package. It addresses problems with respect to the package relationship.

2. There exists no non-static method with the same name as the restored method m in the class containing m , its ancestors, or its descendants. That is, no method will override or overload m , or no method will not be overridden or overloaded by m . This condition addresses problems resulting from the relationship with respect to inheritance and non-static method invocation with polymorphism. A static method can be ignored since it never causes override and overload relationships.
3. There exists no non-static field with the same name as the restored field f in the class containing the field f , its ancestors, or its descendants. That is, no field will not hide f or no field will not be hidden by f . This condition addresses problems resulting from the relationship with respect to inheritance and non-static field access with polymorphism. A static field can be ignored since it never causes a hide relationship.
4. The code using the reflection APIs is not included in the files to be restored. This condition is not serious because source files involving the reflection APIs can not be in general correctly refactored and then undoing such files is rarely required.

If every condition corresponding to the restored element (file, class, method, or field) in the undoable refactoring is satisfied, the target refactoring is truly undoable. In essence, the two conditions 2 and 3 can be relaxed. Restoring methods or fields appropriate for these conditions is feasible as far as all references to them will not be changed. However, the check for such references requires expensive static analysis or run-time analysis. Therefore, the proposed undo mechanism judges by way of the existence of appropriate methods or fields. As a result, the processing time is probably reduced as compared with that when reapplying some of past refactorings.

3.4. Unifying Refactoring and Editorial Changes

Because of a limited kind of automated refactorings, it is inevitable that changes by refactoring and manual editing are mixed. However, conventional undo mechanisms are not much interested in how they undo those changes. For example, the refactoring undo chain (buffer) of Eclipse is only valid as long as no change has been performed after the last refactoring. That is, a programmer cannot execute an undo operation to the refactored source file once it was edited after the refactoring to be undone. Moreover,

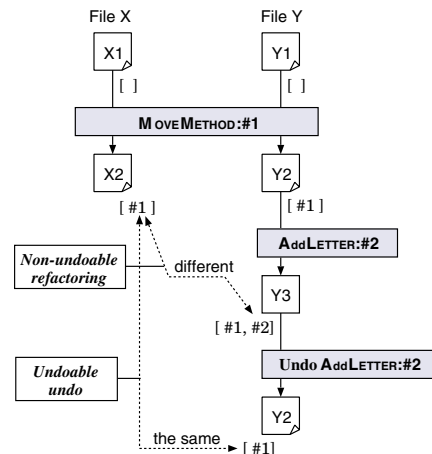


Figure 4. Undo a refactoring involving an editorial action.

the programmer cannot undo any refactoring prior to editorial changes even if the previous edit is canceled. Accordingly, most undo mechanisms are not compatible with their undo/redo facilities of editorial changes.

The proposed undo mechanism aggressively unifies every change of a source file; each change results from either refactoring or manual editing. Here careful readers are aware of the fact that the new undo mechanism works well without modification. This is basically true but little modification should be recommended. Consider that a programmer would move a method from a class of a source file X to a source file Y through a MOVEMETHOD refactoring, and then would add a letter (character) to the content of Y. In this situation, carelessly undoing the previous MOVEMETHOD refactoring induces arbitrary discard of the editorial change (the addition of a letter). This result tends to confuse the programmer although it never gives rise to the inconsistency between the restored contents X1 and Y1.

To avoid this confusion, the proposed undo mechanism assigns identification numbers to not only refactorings but also editorial actions. The identification numbers for editorial actions will be also added to the refactoring undo chain for the edited source file. The mechanism monitors the last of both the applied refactorings and editorial actions. Moreover, it will be either removed from or re-added to the chain depending on a build-in undo or redo operation. Consequently, it forbids a programmer to undo a refactoring if any editorial action was performed after the refactoring. They can undo this refactoring if all the editorial actions are canceled by editorial undo. In Figure 4, for example, the MOVEMETHOD:#1 refactoring cannot be undone after adding a letter to the content of Y. It will become undoable after the past addition of the letter is canceled, that is, the content is restored to Y2 by reversing the editorial change ADDLETTER:#2. With the proposed undo mechanism, the

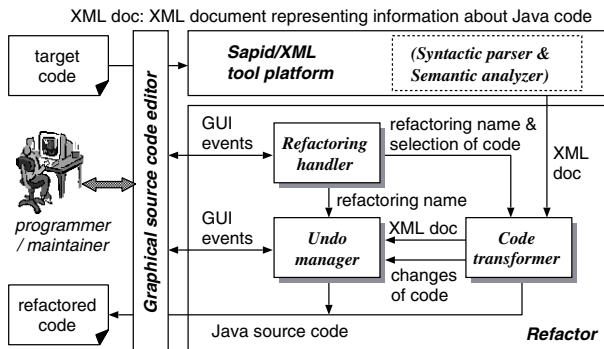


Figure 5. Architecture of Jrbx.

programmer can freely apply both refactoring undo and editorial undo/redo integrated in a standard editor.

4. Implementation

The proposed undo mechanism has been already integrated in our developed refactoring tool, Jrbx (Java refactoring browser with XML) [7]¹. This section describes several components of Jrbx related to the undo mechanism and its running implementation.

4.1. Jrbx: Java Refactoring Browser

Jrbx supports the application of 22 refactoring transformations which can be undone by the proposed undo mechanism. An overall architecture of Jrbx is shown in Figure 5. Jrbx consists of three components: the Sapid/XML tool platform (in short Sapid/XML) [6], a graphical source code editor, and a *Refactor*. Sapid/XML includes a syntactic parser and a semantic analyzer, and then provides APIs for converting Java source code into an XML document (a document of the XSDML representation [6]) and retrieving an XML document corresponding to source code of interest. *Refactor* is a core component of Jrbx, which restructures existing code without altering its behavior.

In Figure 5, the undo manager of *Refactor* and the graphical editor are mainly related to the undo facilities. The undo manager receives the name of an applied refactoring and changes of the target source code (actually a collection of the changed files). It manages chains of past refactorings and sets of files changed together, and determines which refactoring is undoable so that a programmer can undo undesired changes safely. This determination is done by monitoring both GUI events (mouse or keyboard events) arising from programmer's instructions on the editor and undoable refactorings calculated based on the recorded chains. Moreover, the four conditions mentioned in Section 3.3 are all checked by using a fine-grained XML representation of syntactic and semantic information provided by Sapid/XML.

¹Jrbx can be downloaded from <http://www.jtool.org/>.

The graphical editor provides the capabilities to file and edit source code as well as a usual editor. In addition, it activates or deactivates an undo command for an undoable refactoring or a non-undoable one, respectively.

4.2. Design of Undo Manager

Figure 6 shows the design of the undo manager and its related modules. In the development, the Sun Microsystems J2SDK 1.4.2 has been utilized. The class *CodePane* implements each text area containing a Java source file which a programmer is editing. The class *CodesPane* indicates a collection of text areas.

The class *Refactor* executes a specified refactoring by using the method *execute()*. This method creates an instance from the class *RefactoringRecord* for each of the changed files, which represents information about the applied refactoring (the name of the refactoring and a set of the changed files). It stores changes (edits) of each source file by calling the methods *beginCompoundEdit()* and *endCompoundEdit()* of the class *RefactoringUndoManager*. The method *canUndo()* returns the name of the applied refactoring if it can be undone, otherwise *null*. The method *undo()* checks the conditions for implicit relationships by using the method *check()* of the utility class *UndoChecker*, and makes the refactoring undone by calling the method *undoRefactoring()* of *RefactoringUndoManager*.

The class *CodeUndoManager* extends the class *javax.swing.undo.UndoManager* and implements the interface *javax.swing.event.UndoableEditListener*. All methods defined in *CodeUndoManager* support standard undo/redo facilities of editorial actions. The *RefactoringUndoManager* is specialized to implement refactoring undo. The method *endCompoundEdit()* saves a lump of past edits to the field *compoundEdit* of *CodeUndoManager*, and then appends information about the applied refactoring (an instance *record*) to each liner undo list (an instance collected by the role name *commands*) managed by *RefactoringUndoManager*. The methods *undo()* and *redo()* operate both the undo list and a redo stack (an instance collected by the role name *redoCommands*), and then call the respective overridden methods of *CodeUndoManager*.

The class *RefactoringMenuPane* activates or deactivates an undo command (a field *undoItem*) according to the result of invocation to *canUndo()* of *Refactor*.

5. Related Work

The general implementation to reverse the effects of past transformations was presented by [4]. The basic concept of an undo mechanism using refactoring chains was proposed in the Smalltalk Refactoring Browser [11], and has been introduced into the proposed undo mechanism. Although

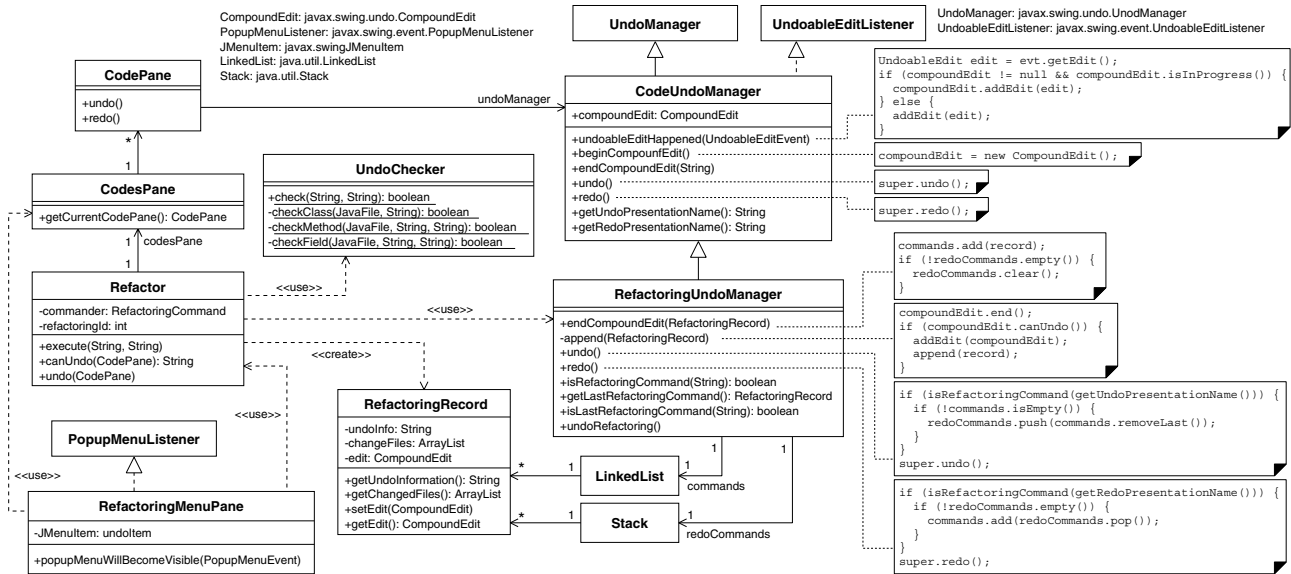


Figure 6. Design related to the undo mechanism.

both the concepts are almost same, the proposed mechanism based on light-weight analysis is much simple since it does not need the troublesome and time-consuming re-application of past refactorings. Moreover, the proposed undo mechanism quits undoing any arbitrary refactoring. Instead, it can undo more refactorings without accidental changes by relaxing the limitation of conventional undo mechanisms, and can be compatible with undoing and redoing editorial changes.

Recently, many refactoring tools support undo mechanisms; nevertheless, their capabilities are limited and they are awkward at handling coincident and consecutive changes of multiple source files. To my knowledge, IntelliJ IDEA [5] is highly capable of undoing several kinds of refactorings, but it does not adapt to the implicit relationships described in Section 3.3. Moreover, it is hard to modify this undo mechanism or embed it into existing or newly-created refactoring tools, since its implementation has been closed. In contrast, the proposed implementation is intended to be freely used.

6. Conclusion

This paper has proposed a new mechanism that can perform accurate and convenient undo of past refactorings, and described the design and implementation of this undo mechanism which was integrated into our developed refactoring browser.

The proposed mechanism will be integrated into a popular IDE such as Eclipse. An extended mechanism not only undoing refactorings but also redoing the undone ones will be developed. Moreover, a way to deal with the deletion and renaming of a source file itself is an issue in the future.

To provide more flexible undo capabilities, the changes of fine-grained program elements (e.g., classes, methods, and fields) instead of files should be separately and hierarchically stored and restored in a source code repository.

References

- [1] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Computer Human Interaction*, 1(3):269–294, 1994.
- [2] Eclipse.org. Eclipse. <http://www.eclipse.org/>.
- [3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [4] J. James E. Archer, R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. Programming Languages and Systems*, 6(1):1–19, 1984.
- [5] JetBrains. IntelliJ IDEA. <http://www.jetbrains.com/idea/>.
- [6] K. Maruyama and S. Yamamoto. A CASE tool platform using an XML representation of Java source code. In *Proc. SCAM'04*, pages 158–167, 2004.
- [7] K. Maruyama and S. Yamamoto. Design and implementation of an extensible and modifiable refactoring tool. In *Proc. IWPC'05*, pages 195–204, 2005.
- [8] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE TSE*, 30(2):126–139, 2004.
- [9] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1992.
- [10] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
- [11] D. B. Roberts. Practical analysis for refactoring. Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1999.