利便性の高いリファクタリングの取り消しメカニズム

丸山勝久[†] 大森隆行[††]

リファクタリングを導入することで，既存ソースコードの外部から見た振舞いを変えずに，理解性や再利用性を改善することが可能である．しかしながら，実際に変換を繰り返し適用してみなければ，リファクタリングによる改善の効果を確認することは難しい．このため，ソフトウェア開発や保守作業を支援する自動化ツールにおいて，リファクタリングの適用とその取り消し機能の提供は不可欠である．本論文では，開発中の個々のソースファイルに対して，過去に適用されたリファクタリングの履歴を鎖で表現し，その鎖の最後に記録されたリファクタリングを監視することで，どのリファクタリングが取り消し可能であるのかを判定するメカニズムを提案する．提案メカニズムは，たとえリファクタリングにより複数のソースファイルが同時かつ連続的に変更されていたとしても，過去のリファクタリングを取り消すことができる．このメカニズムの実装は，我々の開発したリファクタリングツールに組み込まれている．

A Convenient Undo Mechanism for Refactorings

Katsuhisa Maruyama[†] and Takayuki Omori[††]

Refactoring makes existing source code more understandable and reusable without changing external behavior. However, the effect of the improvement cannot be observed without many applications of various kinds of refactorings. Therefore, applying refactorings to existing source code and reversing the effects of past refactorings are inseparable from automated 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 undone although the refactoring has consecutively and/or coincidentally affected multiple source files. A running implementation of the mechanism has been integrated into our developed refactoring tool.

1. はじめに

リファクタリング[1),2)]とは，既存コード（ソースコード）の外部的振舞いを変えずに内部構造を変換することで，そのコードの理解性や再利用性を改善する作業である．このため，近年のソフトウェア開発や保守において，不可欠かつ有用な実践技術と考えられている．また，Eclipse[3)]やIntelliJ IDEA[4)]をはじめとする統合開発環境（IDE：integrated development environments）の多くは，リファクタリングツール（ブラウザ）を備えている．自動化（半自動化）されたリファクタリングツールを用いることで，プログラマは，コード変換にともなう面倒な検査や誤りの混入から完全あるいは部分的に解放される．さらに，リファクタリングにより既存のコードを実際に変換することで，外部的振舞いが同じで内容の異なる2つのコードを，プログラマは見ることができる．このことは，他人によって記述されたコードを理解する助けになる．

しかしながら，リファクタリングによる改善の効果は，適用する変換の種類や適用順序に依存する．このため，実際に変換を繰り返し適用してみなければ，その効果を確認することは困難である．さらには，自動化されたすべてのリファクタリングが，コードの外部的振舞いを保存することを保証することはできない．なぜなら，コードの振舞いが正確に定義されることは珍しく，定義されたとしても実際の変換において活用することが難しいからである[5)]．

効果的なリファクタリングの適用順序が事前に確定できない，さらに，外部的振舞いの保存が保証できないという理由から，リファクタリングツールには，過

† 立命館大学情報理工学部情報システム学科
Department of Computer Science, Ritsumeikan University
†† 立命館大学理工学研究科
Graduate School of Science and Engineering, Ritsumeikan University

去に適用した望まない変換や間違った変換を容易に取り消す（Undo）メカニズムが不可欠である[6]．ここで，もしリファクタリングの適用による変更の影響がつねに単一のソースファイルに閉じ込められているのであれば，その変更を取り消すメカニズムを実現することは容易である．しかしながら，多くのリファクタリングは，同時に複数のソースファイルを書き換える．それにもかかわらず，複数のソースファイルにおいて，同時および連続して発生した変更を，利便性を犠牲にせずに，矛盾なく取り消し可能なリファクタリングツールはいまだ存在しない．ここで，利便性を犠牲にしないとは，最後のリファクタリングだけでなく途中のリファクタリングを取り消し可能であること，および，プログラマによる編集操作を制限せずにリファクタリングが取り消し可能であることを指す．また，矛盾がないとは，リファクタリングの取り消しによりソースコードの外部的挙動が変化しないことを指す．

本論文では，リファクタリングの適用履歴を適切に管理することにより，複数のソースファイルに散らばったリファクタリングによる変更を，矛盾なく取り消すメカニズムを提案する．このメカニズムでは，過去に適用したすべてのリファクタリングに対して識別番号を割り当て，各ソースファイルに対して，リファクタリングの適用順序に基づく識別番号の鎖を作成する．それぞれの鎖の最後に記録された識別番号を監視および比較することで，取り消し可能なリファクタリングを決定する．このようにすることで，取り消しにより影響を受けるソースファイル群において最後に適用されたリファクタリングであれば，それが全体としては最後に適用されたものでなくても取り消しが可能となり，利便性は高くなる．また，提案メカニズムの実装は，通常のソースコードエディタが提供する編集に対する取り消しや取り消しのやり直し（Undo & Redo）との共存を考慮して行われている．よって，リファクタリングによる自動変換とプログラマによる手動編集を，矛盾なく取り消すことができる．提案するリファクタリングの取り消しを実現することで，プログラマは気軽に自動化リファクタリングの適用を試すことができる．さらに，リファクタリングの適用と取り消しを何度でも自由に実行することで，改善の効果を容易に確認することができる．

本論文の構成は，以下のとおりである．2 章では，一般的なリファクタリングプロセスを示し，リファクタリングの取り消しにより発生する問題と従来の解決手法を説明する．次に，3 章では，リファクタリング履歴の鎖を用いた取り消しメカニズムを提案し，4 章で，その実装を簡単に説明する．5 章では，関連研究を紹介する．最後に，6 章でまとめと今後の課題を述べる．

2. リファクタリングの取り消し

本章では，取り消し操作を含むリファクタリングプロセスを示し，取り消しにより発生する典型的な問題を説明する．さらに，この問題を解決するための従来手法と残された課題を示す．

2.1 リファクタリングプロセス

コードの外部的挙動を保存するために，リファクタリングでは大きな変更を，挙動を保存する小さな変換の連続的な適用により実現する．リファクタリングを適用する際のプロセスは，以下のようになる[5]．

手順 1 コード内の不吉な匂い[2]（改善が必要かもしれない兆候）を発見し，リファクタリングを適用する場所を特定する．

手順 2 特定したコード断片に対して，適用するリファクタリングを決定する．

手順 3 決定したリファクタリングを適用する前に満たしていなければならない前提条件を検査する．

手順 4 小さな変換を繰り返すことで，リファクタリングを実行する．もし匂いが部分的に残っていたら，手順 2 に戻る．

手順 5 改善後のソースコードに対する品質特性（たとえば，ISO9126 の保守性，モジュール分割における結合度や凝縮度など）に関して，リファクタリングの効果を評価する．

手順 6 変換後のコードとそれに付随する成果物との間の整合性を維持する．

たとえリファクタリングを手動で行うことが可能だとしても，実際の開発では，自動化ツールによる支援は必須である．このため，たいていの統合開発環境では，手順 3 の前提条件の検査作業と手順 4 のコード書き換え作業を自動化したリファクタリング機能を提供している☆．本論文では，自動化ツールによって実行されたリファクタリングを取り消しの対象とする．

取り消し操作は，通常，手順 4 と手順 5 において，プログラマにより実行される．手順 4 の直後において，適用したリファクタリングが匂いを除去するのに不適当である（と感じた）場合，プログラマは，直前のリファクタリングを取り消す．同様に，手順 5 の直後において，期待した効果が得られなかった，あるい

☆ 手順 3 および手順 4 以外の作業の自動化についても研究が行われているが，現時点では非常に困難である．

は，品質が低下した（と感じた）場合，プログラマは，過去のリファクタリングを取り消したいと考える．通常，品質特性に関する改善は複数回のリファクタリングの適用により達成するため，手順 5 における取り消しは，直前に適用したものだけが対象となるわけではなく，過去に適用したすべてが対象となる．

2.2 動　　機

あらゆるリファクタリングツールにとって，望まない効果を発生させたリファクタリングを取り消す機能を提供することは重要である．しかしながら，リファクタリングにより複数のソースファイルに散らばった変更の影響を正確に把握し，任意のリファクタリングを妥当なコストで取り消すことは容易ではない．事実，Eclipse や IntelliJ IDEA に組み込まれている取り消し操作は，復帰後のコードの外部的振舞いを変更してしまうことがある．また，多くの既存ツールでは，利便性を犠牲にして，取り消し可能なリファクタリングを大幅に制限することで，取り消しの正しさを保証している．

プログラマの実行した直前の編集操作（文字の挿入，削除，置換，テキストの切り取り，コピー，貼り付けなど）を取り消し，ソースファイル単位で復活させる機能を実現する際には，一般的に，制限線形取り消し（restricted linear undo）[7] が用いられる．この取り消しは，編集操作を順番に線形リストに保存することで実現されている☆．この取り消しメカニズムは単純であるため，多くのソースコードエディタが採用している．

いま，複数のソースファイルに散らばった変更を一括して破棄できるように，制限線形取り消しメカニズムを単純に拡張することを考える．**図 1** に 3 つのソースファイル X，Y，Z の複数の版と，それらのソースファイルに適用したリファクタリングを取り消した結果を示す．Xn，Yn，Zn（n = 1, 2, ...）は，それぞれのソースファイルの内容を表す．

図 1 (a) において，プログラマは，まず X と Y に MoveMethod リファクタリングを適用している（たとえば，メソッド `m()` を，それを定義している X 内のクラスから，それを頻繁に使用している Y 内のクラスに移動している）．さらに，Z に RenameField リファクタリングを適用している（たとえば，Z のクラス内のフィールド `f` を適切な名前に変更している）．このような状況において，プログラマが X に対してリファクタリングの取り消しを実行すると，X に適用された最後のリファクタリング（MoveMethod）が破棄され，X およびそれに関連のある（リファクタリングの適用により同時に変更された）Y の内容を，それぞれ X1 と Y1 に戻す．この取り消しは問題を引き起こさない（Y や Z も問題なく取り消し可能である）．

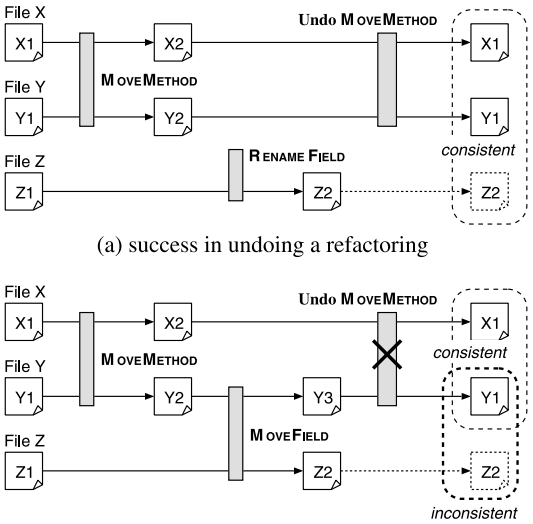


図 1 リファクタリングによるコードの変換とその取り消し
Fig. 1 Undo a refactoring affecting multiple files.

しかしながら，図 1 (b) に示すように，ソースファイル X，Y，Z に適用された異なるリファクタリングが絡まっている（Y を介して干渉している場合），このような単純な取り消しメカニズムでは，これらのソースファイルの内容の無矛盾性を破壊する．つまり，取り消し前の外部的振舞いを保存しないことがある．X に取り消し操作を実行した際，X と Y の内容を X1 と Y1 に戻すと，外部的振舞いを保存するという観点において同時に存在することが許されない Y1 と Z2 という組合せが発生してしまう．たとえば，MoveMethod において `m()` が X から Y に移動しており，MoveField において `f` が Y から Z に移動していた場合，取り消し後の Y1 と Z2 の両方が同時に `f` を定義することになる．あるいは，MoveField において `f` が Z から Y に移動していた場合，`f` は取り消し後の Y1 と Z2 のどちらからも消えてしまう．通常，プログラマはこのような取り消しを許容しない．

2.3 従 来 手 法

2.2 節で述べた問題を回避するため，Roberts は，リファクタリングの取り消しに対して 3 つの実装方式を提案している[8]．1 番目の方式では，逆変換（inverse refactoring）を用いる．このような発想は明確かつ理

☆ 制限線形取り消しでは，新しい編集操作が実行された時点で，取り消しのやり直し（redo）リストを空にすることで，不当なやり直しを回避する．

解しやすい反面，既存のすべてのリファクタリングに対して，逆変換を用意することが困難であるという問題を持つ．このため，取り消し機能の実現に，この方式が利用されることはめったにない．

2 番目の方式では，個々のリファクタリングの適用後にチェックポイントを設定し，コード全体の内容（実際には変更差分）を，リファクタリングの適用順に 1 つの鎖に記録する．取り消しの際には，鎖に記録されたチェックポイントを 1 つ戻すことで，コード全体を直前の内容に戻す．取り消しの順序はリファクタリングの適用順序と必ず逆になるため，取り消し後のコードにおいて内容の無矛盾性は保証される．しかしながら，この方式では，鎖の中間に記録された任意のチェックポイントに対応するリファクタリング，つまり，最後に適用されたリファクタリング以外のものを取り消せないという利便性（使いやすさ）に関する欠点がある．図 1 (a) を用いて説明すると，プログラマは，ファイル Z に対して RenameField を取り消さなければ，ファイル X あるいは Y の MoveMethod を取り消すことはできない☆．このような制限は，多くのプログラマにとって不便である．

このような不便さを解消するために，3 番目の方式では，過去に適用されたリファクタリングの依存関係（干渉）を表現した鎖を用いる．この方式では，まず取り消したいリファクタリングの適用前の内容にソースファイルをいったん戻し，取り消したいリファクタリング後に適用されたすべての変換（実際には，依存関係のある変換のみ）を再適用することで，鎖に記録された任意のリファクタリングの取り消しを実現する．もし再適用できないリファクタリングが存在した場合は，取り消しを中止する．この方式は，実際に Smalltalk Refactoring Browser[8] に組み込まれている．任意のリファクタリングの取り消しを実現している点で，この方式は利用価値が高いように見える．しかしながら，次に示す 3 点から，多くのリファクタリングツールにとって必ずしも適した方式とはいえない．

(1) 再適用を実現する実装の困難さ

リファクタリングを再適用するためには，プログラマが指定したコード断片（の位置）を特定する情報（たとえば，RenameMethod リファクタリングではもとのメソッドの名前）を把握しておかなければならない．リファクタリング前後の 2 つの版（コードのスナップショット）において，クラス，そのメンバ（メソッドやフィールド），メソッドの引数は，それらの名前を用いることで，対応するコード断片を特定することが比較的容易である．しかしながら，もしリファクタリングが（名前を持たない）文や（同一のメソッド内部で繰り返し宣言される可能性がある）ローカル変数を対象とし，それらの位置（オフセット）を変更してしまった場合や，リファクタリングの前後にコードを手動で編集してしまった場合，対応するコード断片を版をまたいで特定することは非常に困難である．よって，このような変更をともなうリファクタリング（たとえば，Fowler のカタログ[2] にある SplitTemporaryVariable）の再適用は不可能なことが多い．また，リファクタリングツールは，通常，コードを変換する際にプログラマと対話する．たとえば，プログラマは，RenameMethod リファクタリングにおいて新規のメソッド名を入力したり，MoveMethod リファクタリングにおいてメソッドの移動先クラスを指定したりする．リファクタリングを再適用するためには，このような対話によって得られた情報をすべて管理しておかなければならない．このように再適用の実装は困難である．

(2) 再適用による予想外の結果

取り消したリファクタリングと再適用するリファクタリングにより変更されるコードに使用（参照）関係が存在する場合，取り消し前後でその内容が大きく異なるコードが生成されることがある．このことはプログラマを混乱させる原因となり，任意のリファクタリングを取り消し可能という利点を放棄するのに十分な理由となる．たとえば，あるメソッドを別のクラスに移動させた（MoveMethod リファクタリング）後に，そのメソッドからそのメソッドから新規にメソッドを抽出した（ExtractMethod リファクタリング）場合を考える．この場合に MoveMethod を取り消すと，メソッドの移動元クラスにおいて ExtractMethod が再適用されることになる．このことは，移動先クラスから 2 つのメソッド（移動メソッドと新規抽出メソッド）が同時に移動先クラスに戻っているように見え，プログラマは混乱する．さらに，新規抽出メソッドを呼び出しているコードが別のクラスに存在していた場合，クラス間の使用関係も変化するため，取り消しによる結果を予想しにくい．

(3) 再適用に費やすコスト

リファクタリングの自動適用は，一般的に，コードに対する広域な静的解析を要求する．このため，再適用には，高いコストが費やされる．たとえば，一般的なリファクタリング操作の実装では，単にメソッドの

☆ Eclipse の 3.0 版までのリファクタリング取り消しでは，最後に適用したリファクタリングしか取り消せない．

名前を変更するだけでも，そのメソッドを呼び出している場所を特定するために，プログラマが記述しているソースコード全体を解析する必要がある．

3. リファクタリングの鎖を用いた取り消し

取り消しにおける利便性を犠牲にせず，さらに，実装が難しい再適用を避けるため，提案する取り消しメカニズムでは，過去に適用したリファクタリングの履歴に基づく鎖を活用する．本章では，取り消し可能リファクタリングを定義し，それを用いた取り消しメカニズムの詳細を述べる．

3.1 最終リファクタリング

本メカニズムでは，ソースファイルごとにその内容の変化を記録するチェックポイントがソースエディタにより提供されており，編集操作の取り消しが実現されていることを前提とする．そのうえで，ソースファイルごとに鎖（線形リスト）を用意する．プログラマがソースファイルにリファクタリングを適用したとき，新たに識別番号（シーケンス番号）を生成し，その識別番号を変更の影響を受けたそれぞれのソースファイルの鎖に追加する．それと同時に，変更されたソースファイルの集合も記録する．記録されたファイル集合は，鎖内の識別番号をキーとして検索することができるようになっている．提案する取り消しメカニズムは，識別番号の鎖と記録されたソースファイル集合に基づき，リファクタリングの取り消し可能性を判断する．最終リファクタリング（last refactoring）を以下のように定義する．

> i 番目のリファクタリング r_i（識別番号 i）により同時に変更されたすべてのソースファイルを要素に持つ集合を S_i とおく．S_i 内のすべてのソースファイルが，それぞれの鎖の最後に同一の識別番号 i を持つ（$\forall s \in S_i[\ last(s) = i\ ]$）とき，$r_i$ を**最終リファクタリング**という．

ここで，$last(s)$ は，ソースファイル s の鎖において，最後に追加された（最後に適用されたリファクタリングの）識別番号を指す．上記の定義を満たさないものを，取り消し不可能リファクタリング（non-undoable refactoring）と呼ぶ．

提案する取り消しメカニズムは，プログラマが編集中のソースファイルに対して，最後に適用されたリファクタリング r_i が上記の条件を満たすかどうかをリファクタリングメニューの表示時に随時検査し，最終リファクタリングの場合にのみ，取り消しコマンド（ソースコードエディタの取り消しメニュー項目）を活性化する．実際に取り消し操作が実行された場合，このメカニズムは S_i 内のそれぞれのソースファイルに対して，リファクタリングによる編集操作を一括して取り消すことで，それらの内容を同時に r_i の適用前（各ファイルにおける直前の内容）に戻す．その後，それらのソースファイルの鎖から最後尾の識別番号を取り除く．もし r_i が取り消し不可能であるときは，プログラマが取り消しを実行できないように，取り消しコマンドは非活性になる．

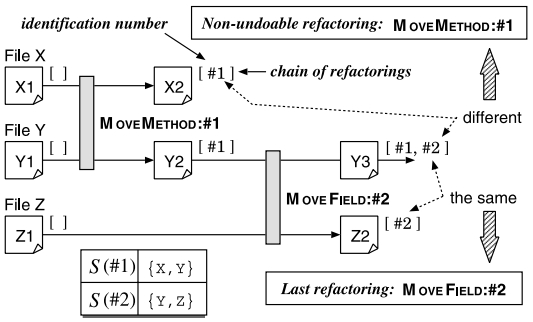


図 2 最終リファクタリングおよび取り消し不可能リファクタリング
Fig. 2 Last refactoring and non-undoable refactorings.

図 1 (b) に示したコードに対して，最終リファクタリングおよび取り消し不可能リファクタリングを**図 2** に示す．各リファクタリングに付与されている記号#1 と#2 は，その識別番号を指す．いま，Y および Z の鎖に最後に存在する識別番号はどちらも#2 であるため，Y あるいは Z の編集中において，プログラマは MoveField:#2 リファクタリングを取り消すことができる．一方，X と Y の鎖の最後に存在する識別番号は異なる（それぞれ#1 と#2 である）ため，X の編集中において，MoveMethod:#1 リファクタリングを取り消すことはできない．もし，MoveField:#2 が事前に取り消されれば，MoveMethod:#1 の取り消しは可能となる．

各ソースファイルの鎖に記録される識別番号の順序，つまり，リファクタリングの適用順序は，コード全体におけるリファクタリングの適用順序と矛盾しない．つまり，識別番号の鎖を列（sequence）と見なした場合，各ファイルの列はコード全体の列の部分列（subsequence）となる．よって，個々のソースファイル単位で見ると，このメカニズムによる取り消しの順序は必ず過去のリファクタリングの適用順序と逆になる．

3.2 ソースファイル間の明示的関係と暗黙的関係

提案メカニズムは，取り消し可能リファクタリングの定義に基づき，互いに関係を持つソースファイル群を単位として，過去のリファクタリングを取り消す．

この取り消しがコードの外部的振舞いを保存するためには，ソースファイル間に存在する関係を慎重に扱う必要がある．ここでは，リファクタリングの対象をJavaプログラムとし，次に示す7つの特性により，異なるJavaソースファイル間の関係を定義する．

(1) パッケージ宣言によるクラスの含有（**package** 節を用いることで，異なるファイルで定義されたクラスが同じパッケージに属することができる）
(2) インポート宣言によるクラスの取り込み（**import** 節を用いることで，異なるパッケージに存在する別のファイルで定義されたクラスを，単純名によって参照することができる）
(3) 継承（インタフェース実装を含む）
(4) メソッド呼び出し
(5) フィールド参照
(6) クラスの型参照（インスタンス生成を含む）
(7) Reflection API の利用（リフレクション）

提案する取り消しメカニズムでは，ソースファイルの一部ではなく，ファイル全体の内容を一括して復帰させる．このため，単一のソールファイルに閉じ込められる関係は，そのままコード内に残るか同時に消去されることになり，ここでは無視できる．

上記の7つの特性による関係は，明示的関係と暗黙的関係に分けられる．明示的関係とは，リファクタリングにより，必ず同時に内容が書き換えられるソースファイル間に成立する関係を指す．一方，リファクタリングを適用したソースファイルに対して暗黙的関係を持つソースファイルの内容は，必ずしも同時に書き換えられるとは限らない．つまり，適用したリファクタリングにより内容が書き換えられたソースファイルと関係を持つにもかかわらず，その識別番号が鎖に記録されないソースファイルが存在する．特性(2)の単一の型インポート宣言，特性(6)の型参照は明示的関係に分類される．残りの関係，つまり，特性(1)のパッケージ宣言，特性(2)のオンデマンドの型インポート宣言，特性(3)の継承，特性(4)のメソッド呼び出し，特性(5)のフィールド参照，特性(7)のリフレクションが，暗黙的関係に分類される．

明示的関係だけを持つ（暗黙的関係を含まない）ソースファイルに対するリファクタリングの取り消しは，複数のファイルに対して，それらの内容を復帰させた場合に決して矛盾を発生させない．なぜなら，リファクタリングが適用されたソースファイルに対して明示的関係を持つすべてのソースファイル群には，そのリファクタリングを指す識別番号が必ず記録されているからである．このリファクタリングが取り消される場合は，これらのソースファイル群に対する書き換えは必ず同時に破棄される．よって，取り消したいリファクタリングがコードの挙動を保存しており，それにより影響を受けるすべてのソースファイルの鎖にその識別子が明示的に記録されている限り，取消しによりコードの挙動が変わることはない（付録 A.1 参照）．

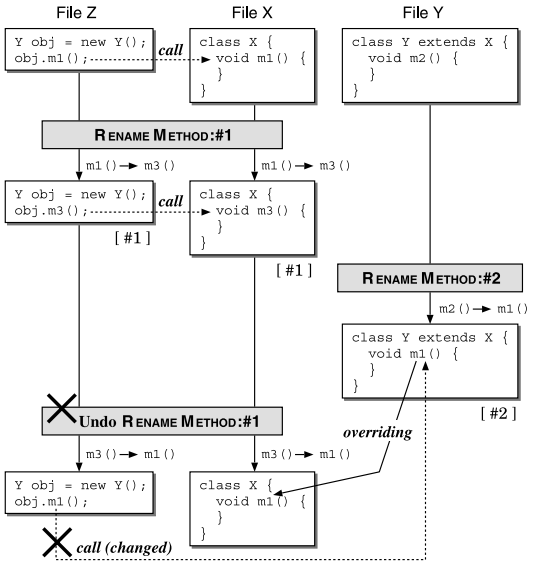


図 3 暗黙的関係を含むリファクタリングの取り消し
Fig. 3 Undo a refactoring involving implicit relationships.

しかしながら，暗黙的関係を含むリファクタリングの取り消しは問題を引き起こす．たとえば，**図 3** に示す3つのソースファイルを X，Y，Z を考える．これらのファイル群は，継承と（多相的）メソッド呼び出しという暗黙的関係を含んでいる．具体的には，クラス Y がクラス X の子クラス，つまり，Y は X の特性（初期状態におけるメソッド m1() の内容）を引き継いでおり，Z が，クラス Y のインスタンス通して，クラス X のメソッド m1() を呼び出している．

いま，プログラマが，ファイル X に対して RenameMethod:#1 リファクタリングを適用することで，クラス X のメソッド m1() の名前を m3() に変更し，さらに，ファイル Y に対して RenameMethod:#2 リファクタリングを適用することで，クラス Y のメソッド m2() の名前を m1() に変更した場合を考える．これらのリファクタリングは，3つのソースファイルのコードの挙動を保存している．この状況において，もしプログラマが RenameMethod:#1 リファクタリングを取り消した場合，クラス X と Y の両方が同じ名前（シグニチャ）のメソッド m1() を同時に持つことになり，これらのメソッドはオーバライドの関係を有する．結果として，ファイル Z から実際に呼び出

されるメソッドが，クラス X の m1() からクラス Y の m1() に変化するため，取り消し操作を実行する前のコードの挙動は，取り消し後において保存されなくなる．

3.3 取り消し可能リファクタリング

暗黙的関係により発生する問題を解決するためには，2つの方法が考えられる．1つ目の方法では，リファクタリングを適用した際に，すべての暗黙的関係を明示的関係に変換する．たとえば，RenameMethod リファクタリングの適用において，名前が変更されたメソッドを含むクラスの祖先クラスと子孫クラスを定義しているすべてのファイル群に，リファクタリングの識別番号を記録することで，継承とメソッド呼び出しによる暗黙的関係を明示的関係として扱うことができる．同様に，あるクラスを別のパッケージに移動させる MoveClass リファクタリングの適用では，移動したクラス，移動元パッケージ内のすべてのクラス，移動先パッケージ内のすべてのクラスを定義しているファイル群に，識別番号を記録する．この解決方法は直感的であるが，不必要に多くの明示的関係を生成してしまうため，取り消し可能なリファクタリングの多くを取り消し不可能リファクタリングと判定するという重大な欠陥を持つ．

もう1つの解決方法では，リファクタリングの取り消しが，挙動を変化させないことを保証するために，いくつかの条件を用意する．これらの条件は，取り消し実行直前に検査される．この方法では，取り消したいリファクタリングに応じて条件の検査が行われるため，1つの目の方法とは異なり，不必要に多くのリファクタリングを取り消し不可能と判定することはない．このような理由から，提案する取り消しメカニズムでは，2つ目の解決方法を採用し，3.1 節で述べた最終リファクタリングに対して，以下に示す3つの取り消し実行条件（続行条件）を付加する．

条件 1 取り消しにより復活するクラス c と同一の名前を持つクラスが，c を含むパッケージ内に存在しない．この条件は，同一の名前を持つ複数のクラスが1つのパッケージ内に存在できないという Java の言語仕様に起因し，特性 (1) のパッケージ宣言，あるいは，特性 (2) のオンデマンドの型インポート宣言の暗黙的関係により発生する問題に対処する．

条件 2 取り消しにより復活するメソッド m と同一の名前を持つメソッドが，m を含むクラス，その祖先クラス，および子孫クラスに存在しない．すなわち，m をオーバライドやオーバロードするメソッドが存在しない，あるいは，m にオーバライドやオーバロードされるメソッドが存在しない．同一のシグニチャではなく同一の名前を持つメソッドを検査対象とし，オーバライド関係だけでなく，オーバロード関係も回避している理由は，取り消し前後でオーバロード関係がオーバライド関係に変わり，メソッドの呼び出し先が変化する可能性をなくすためである．引数の追加リファクタリング（AddParameter）や引数の削除リファクタリングの適用がこれに該当する．この条件は，特性 (3) の継承と特性 (4) のメソッド呼び出しにより発生する問題に対処する．

条件 3 取消しにより復活するフィールド f と同一の名前を持つフィールドが，f を含むクラス，その祖先クラス，および子孫クラスに存在しない．すなわち，f を隠蔽，あるいは，f に隠蔽されるフィールドが存在しない．この条件は，特性 (3) の継承と特性 (5) のフィールド参照により発生する問題に対処する．

一般的に，Reflection API を含むソースファイルのリファクタリングにおいて挙動の保存を保証することは困難であるため，このようなファイルに対するリファクタリングはめったに行われない．よって，特性 (7) のリフレクションについては，提案メカニズムで取り扱わない．

3.1 節の最終リファクタリングにおいて，復活する構文要素（ファイル，クラス，メソッド，フィールド）に対応する上記のすべての条件が満たされる場合，そのリファクタリングはまさに取り消し可能（undoable refactoring）となる．実際には，復活するメソッドに対する呼び出しやフィールドに対する参照が変化しない限り挙動の保存に関する問題は発生しないため，条件2と条件3を緩めることが可能である．しかしながら，そのような呼び出しや参照を検査することは，さらなる静的解析や動的解析を要求する．よって，提案メカニズムでは，軽量な解析により実現可能なクラス，メソッド，フィールドの存在だけにより，取り消し実行の妥当性を判断する．このような解法を採用することにより，リファクタリングの再適用に比べて，処理時間を大幅に削減することできる．

3.4 リファクタリングと編集操作の統合

自動化リファクタリングの種類は限られているため，実際の開発では，あらかじめ用意された変換だけで，プログラマの望むコード改善が必ずしも達成されるわけではない．よって，リファクタリングと手動による編集操作の混合は避けられない．しかしながら，従来の取り消しメカニズムは，このようなコードの変化を

どのように扱うのかに関心がない☆.

提案メカニズムでは，リファクタリングによるソースファイルへの変更を 1 つの複合編集操作として扱い，リファクタリング取り消し専用の線形リストに加えて，ソースコードエディタの持つ取り消し用の線形リストにも挿入する．つまり，ソースコードエディタの取り消し機能の観点からは，通常の編集とリファクタリングによる変更は区別されず，一元的に管理される．実際にリファクタリングを取り消す場合には，そのリファクタリングが取り消し可能であるかと同時に，ソースコードエディタの持つ取り消し用線形リストの最後がリファクタリングによる変更操作であるかどうかを検査する．このようにすることで，あるソースファイルに対して，リファクタリング（たとえば，MoveMethod:#1）を適用した後に編集操作（たとえば，文字 “a” の挿入）が実行されている場合，そのリファクタリングおよびリファクタリングの適用後に実行された編集操作を不用意に取り消すことはできなくなる（実際の実装では，リファクタリングの取り消しと同時に編集操作を破棄してもよいかどうかを確認するメッセージが表示される）．リファクタリング後に実行されたすべての編集操作（この例では，文字 “a” の挿入）が取り消された場合，ソースコードエディタの持つ取り消し用の線形リストの最後がリファクタリング（この例では，MoveMethod:#1）による変更となるため，それを取り消すことが可能である．

4. 実　　装

提案メカニズムの実装は，我々の開発したリファクタリングツール Jrbx（Java Refactoring browser with XML）[9] に組み込まれている．Jrbx は，22 個のリファクタリング操作の自動化を実現しており，それらはどれも提案メカニズムを用いて取り消し可能である．

図 4 に，取り消し機能に関連する設計の一部を示す☆☆．開発においては，Sun Microsystems の J2SDK 1.4.2 を利用した．

クラス CodePane は，プログラマが編集中の Java ソースファイルを含むテキストエリアを実装している．

クラス Refactor は，指定されたリファクタリングをメソッド `execute()` により実行する．このメソッドは，変更された各ファイルごとに，クラス RefactoringRecord のインスタンスを生成する．このインスタンスには，適用されたリファクタリングに関する情報（リファクタリングの名前と，同時に変更されたファイル集合）が格納される．さらに，Refactor は，クラス CodeUndoManager のメソッド `beginCompoundEdit()` とクラス RefactoringUndoManager のメソッド `endCompoundEdit()` を呼び出すことで，個々のソースファイルに対するリファクタリングによる変更を一括して保管する．メソッド `canUndo()` は，指定したリファクタリングが取り消し可能であった場合にはその名前を，さもなければ `null` を返す．メソッド `undo()` は，ユーティリティクラス UndoChecker のメソッド `check()` を用いることで，3.3 節で述べた暗黙的関係に対処する 3 つの条件を検査し，RefactoringUndoManager のメソッド `undoRefactoring()` を呼び出すことで，実際にリファクタリングを取り消す．`checkClass()`，`checkMethod()`，`checkField()` の引数に現れるクラス `JavaFile` は，Sapid/XML tool platform [10] により提供される構文情報と意味情報を保持する．

CodeUndoManager において定義されるすべてのメソッドは，編集操作に対する標準的な取り消し機能や取り消しのやり直し機能を提供する．RefactoringUndoManager は，リファクタリングの取り消しを実装するように CodeUndoManager を特化したものである．メソッド `endCompoundEdit()` は，リファクタリングによる複合変更を CodeUndoManager のフィールド `compoundEdit` に保存する．その後，メソッド `append()` を呼び出すことで，RefactoringUndoManager によって管理されている線形リスト（ロール名 `commands` により集められているインスタンス）に対して，適用されたリファクタリングに関する情報（インスタンス `record`）を追加する．

クラス RefactoringMenuPane は，Refactor のメソッド `canUndo()` を呼び出した結果に応じて，取り消しコマンド（フィールド `undoItem`）を活性化あるいは非活性化する．

5. 関 連 研 究

対話的なシステムの構築において，過去の操作を取り消す機能を提供することは重要である．このため，単純な線形取り消しモデルのほかに，数多くの非線形取り消しモデルが提唱されている [7),11)~18)]．これらの取り消しモデルでは，過去の任意の操作を取り消したり，複数の利用者の共同作業における操作を取り消

☆ たとえば，Eclipse（3.0 およびそれ以前の版）では，リファクタリングの適用後に一度でも編集操作が行われていると，たとえ編集操作が取り消されても，そのリファクタリングを取り消すことはできない．

☆☆ メッセージの表示など GUI に関する部分は省略してある．また，最新の実装では，リファクタリングに関する取り消しのやり直しの実装も試みているが，これに関する部分も省略した．

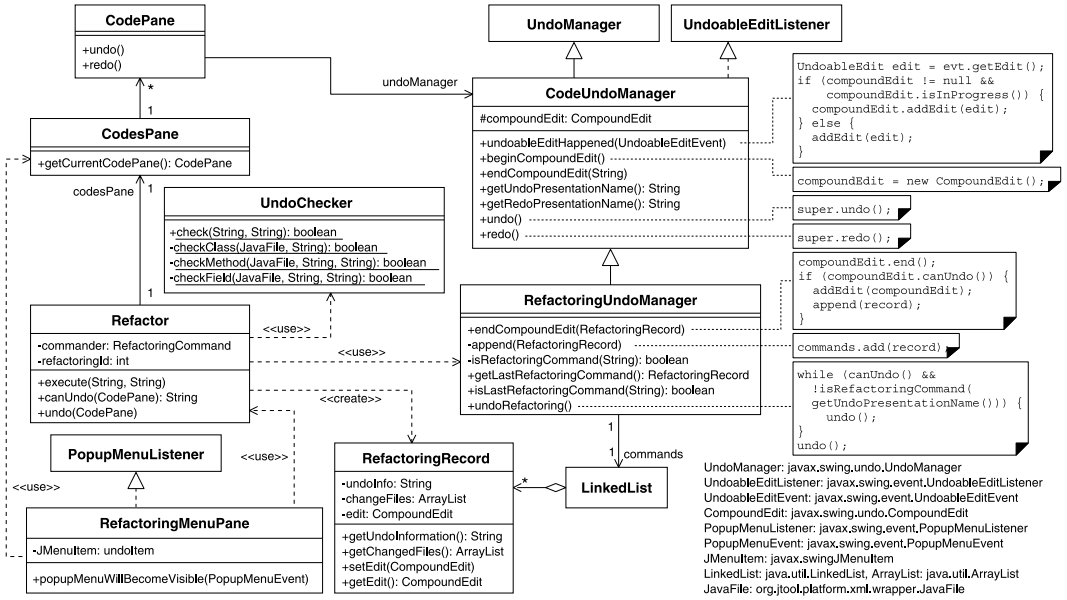


図 4 取り消しメカニズムの設計
Fig. 4 Design related to the undo mechanism.

したりできる．本論文で提案している取り消しメカニズムは，各ソースファイル単位で見ると，単なる線形取り消しの拡張（リファクタリング履歴に基づく制約が付加された線形取り消し）であるが，コード全体で見ると，文献 7) の選択的取り消しの一種であるといえる．

従来の取り消しメカニズムと提案メカニズムとの大きな違いは，取り消し対象である操作の特性である．一般的に，ソースコードエディタや図形エディタにおいて，利用者による手動による単独の操作が同時に複数のファイルを書き換えることはない．これに対して，自動リファクタリングの適用においては，利用者（プログラマ）が単一にもかかわらず，単独の操作が複数のファイルを同時に書き換える．また，プログラミング環境に特化すると，通常のソースコードエディタはファイルごとに独立した線形取り消しリストを備えている．このように取り消し対象として想定している操作や環境が異なるため，従来の取り消しメカニズムを単純に適用するだけでは，利便性の高いリファクタリングの取り消し機能を実現することはできない．

過去に適用したリファクタリングの鎖を取り消しに利用するという着想は，Smalltalk Refactoring Browser (SRB)[8] にすでに取り込まれている☆．提案メカニズムにおけるリファクタリングの鎖の利用は，この取り消しメカニズムに基づいている．たとえ，これらのメカニズムの着想に共通部分が多く見られたとしても，軽量な静的解析に基づく提案メカニズムは，任意のリファクタリングを取り消す手段をあえて放棄することで，過去のリファクタリングの再適用を必要としない．このため，実装の簡潔さ（容易さ），および，実行時間の面で有利である．また，プログラマとっての予想外の変更を抑え，コードが大きく変化することによる混乱を避けている．ここで，SRB に対して，提案メカニズムと同様に，「同時に変更されたソースファイル群の最後のリファクタリングのみを取り消し可能とする」という制限を付加することを考えてみる．この場合，2.3 節の (2) で述べた再適用による予想外の結果の発生を回避することができる．さらに，各リファクタリングを適用する際に，同時に変更されたソースファイル群を適切に管理することで，2.3 節の (1) で述べた実装の困難な再適用や，(3) で述べた広域な静的解析も避けることができる可能性は高い．このように考えると，提案メカニズムは新たなものではなく，SRB の取り消し機能を制限し，さらに，(Java 言語の特性に応じて）同時変更ファイル群を管理するように SRB を拡張したものであるといえる．

近年，多くのリファクタリングツールが取り消し機能を提供している．それにもかかわらず，それらの能力は限られており，複数のソースファイルにまたがる同時および連続的変更をうまく扱えない．たとえば，Eclipse（3.0 およびそれ以前の版）では，最後に適用

☆ この取り消しメカニズムは，文献 11) のスクリプトモデルに類似している．

したリファクタリングしか取り消せない，編集操作との協調を扱えないなど不便な点が多い．このような問題を解決するために，Eclipse 3.1 以降の版では，リファクタリングの取り消しメニューが通常の編集取り消しメニューに統合されたが，リファクタリングと編集操作との協調に欠陥があり，リファクタリングが取り消せなくなったり，取り消し後にコードの外部的挙動が保存されない場面が頻繁に発生する．我々の知る限り，IntelliJ IDEA [4] が，さまざまな種類のリファクタリングを取り消すことができるという点で高機能かつ信頼性が高い．しかしながら，このツールも 3.3 節で述べた暗黙的関係の取り消しには対応していない．また，我々のリファクタリングツールと異なり，実装が非公開であるため，その取り消しメカニズムを修正したり，既存のツールあるいは新規のツールに組み入れたりすることは不可能に近い．

6. おわりに

本論文では，複数のソースファイルを同時に変更するリファクタリングを取り消す利便性の高いメカニズムを提案し，その実装を述べた．残念ながら，現時点で，提案メカニズムによる取り消し操作が真に正しいことは証明されていない．これを証明するためには，3.2 節で述べたソースファイル間の関係の網羅性を十分に調査し，取り消しの中止を判断する条件を検証する必要がある．また，現在，リファクタリングの取り消しだけでなく，取り消したリファクタリングのやり直しを実現するために，提案メカニズムを拡張中である．さらに，ソースファイル自体の削除や名前変更をどのように扱うかに関しては，今後の課題である．

変更の有無をファイル単位で管理すると，実際には異なる 2 つの変更が干渉していないにもかかわらず，干渉が発生すると判断され，取り消しが抑制されることがある．さらに利便性の高い（柔軟な）取り消しメカニズムを提供するためには，プログラムの保存単位をソースファイルではなく，クラス，メソッド，フィールドのように細粒度かつ階層的に扱い，変更影響解析手法[19] などを積極的に活用して，変更を管理することが考えられる．

最後に，本論文で提案する取り消しメカニズムを備えた Jrbx は，`http://www.jtool.org/` からダウンロード可能である．

参考文献

1) Opdyke, W.F.: Refactoring Object-Oriented Frameworks, Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign (1992).
2) Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999). 児玉公信，友野昌夫，平澤　章，梅澤真史（訳）：リファクタリング，ピアソン・エデュケーション (2000).
3) Eclipse.org: Eclipse. http://www.eclipse.org/
4) JetBrains: IntelliJ IDEA. http://www.jetbrains.com/idea/
5) Mens, T. and Tourwé, T.: A Survey of Software Refactoring, *IEEE Trans. Softw. Eng.*, Vol.30, No.2, pp.126–139 (2004).
6) Roberts, D., Brant, J. and Johnson, R.: A Refactoring Tool for Smalltalk, *Theory and Practice of Object Systems* (*TAPOS*), Vol.3, No.4, pp.253–263 (1997).
7) Berlage, T.: A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects, *ACM Trans. Computer Human Interaction*, Vol.1, No.3, pp.269–294 (1994).
8) Roberts, D.B.: Practical Analysis for Refactoring, Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign (1999).
9) Maruyama, K. and Yamamoto, S.: Design and Implementation of an Extensible and Modifiable Refactoring Tool, *Proc. Int'l Workshop Program Comprehension* (*IWPC*), pp.195–204 (2005).
10) Maruyama, K. and Yamamoto, S.: A CASE Tool Platform Using an XML Representation of Java Source Code, *Proc. Workshop Source Code Analysis and Manupilation* (*SCAM*), pp.158–167 (2004).
11) Archer, J.J.E., Conway, R. and Schneider, F.B.: User Recovery and Reversal in Interactive Systems, *ACM Trans. Prog. Lang. and Syst.* (*TOPLAS*), Vol.6, No.1, pp.1–19 (1984).
12) Vitter, J.S.: US & R: A New Framework for Redoing, *IEEE Software*, Vol.1, No.4, pp.39–52 (1984).
13) Prakash, A. and Knister, M.J.: A Framework for Undoing Actions in Collaborative Systems, *ACM Trans. Computer-Human Interaction* (*TOCHI*), Vol.1, No.4, pp.295–330 (1994).
14) Choudhary, R. and Dewan, P.: A General Multi-User Undo/Redo Model, *Proc. European Conf. Computer Supported Work* (*ECSCW*), pp.231–246 (1995).
15) Ressel, M. and Gunzenhäuser, R.: Reducing the Problems of Group Undo, *Proc. Int'l ACM SIGGROUP Conf. Supporting Group Work* (*GROUP*), pp.131–139 (1999).
16) Chen, D. and Sun, C.: Undoing Any Oper-

ation in Collaborative Graphics Editing Systems, *Proc. Int'l ACM SIGGROUP Conf. Supporting Group Work* (*GROUP*), pp.197–206 (2001).
17) Sun, C.: Undo as Concurrent Inverse in Group Editors, *ACM Trans. Computer-Human Interaction* (*TOCHI*), Vol.9, No.4, pp.309–361 (2002).
18) Washizaki, H. and Fukazawa, Y.: Dynamic Hierarchical Undo Facility in a Fine-Grained Component Environment, *Proc. Int'l Conf. Technology of Object-Oriented Languages and Systems* (*TOOLS Pacific 2002*), pp.191–199 (2002).
19) Ren, X., Shah, F., Tip, F., Ryder, B.G. and Chesley, O.: Chianti: A Tool for Change Impact Analysis of Java Programs, *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA*), pp.432–448 (2004).

付　　録

A.1 取り消しが挙動を保存することの説明

ソースコードに対して，2つのリファクタリング ra, rb をこの順番に適用した場合を考える．これら2つが独立（無干渉，非依存）のとき，それぞれのリファクタリングにより影響を受けるソースファイル群は独立（ra および rb により影響を受けるソースファイルの集合を S_{ra} および S_{rb} とすると，$S_{ra} \cap S_{sb} = \emptyset$）となる．この場合，リファクタリングの適用順序を rb, ra と変えても，2つのリファクタリングの適用後のソースコードは同一である．適用されたリファクタリングが外部的挙動を保存する限り，コード全体に対して最後に適用されたリファクタリングを取り消しても，外部的挙動は変化しない．以上より，ra あるいは rb のどちらを先に取り消しても，コードの外部的挙動は保存される．このことは，ra を独立なリファクタリング ra_1 と ra_2 に分割しても成立するので，分割を繰り返すことで，任意の数の独立なリファクタリングに関して成立する．

提案メカニズムでは，ra と rb が独立でない（干渉がある）とき，つまり，$S_{ra} \cap S_{rb} \neq \emptyset$ のとき，ra を rb より前に取り消すことはできない．よって，この場合の取り消しは必ず rb, ra の順番となり，これはコード全体で見た際のリファクタリングの適用順序と逆であるため，外部的挙動は保存される．

(平成18年11月30日受付)
(平成19年 3 月 1 日採録)



丸山　勝久（正会員）

1991年早稲田大学理工学部電気工学科卒業．1993年同大学院理工学研究科修士課程修了．同年日本電信電話株式会社入社．2000年4月より立命館大学理工学部助教授．2004年4月より同大学情報理工学部助教授．2007年4月より同学部教授．2003年9月〜2004年9月 University of California, Irvine 客員研究員．ソフトウェア保守，プログラム解析，ソフトウェア開発環境の研究に従事．情報処理学会1997年度山下記念研究賞，1997年度前期全国大会奨励賞授賞．博士（情報科学）．電子情報通信学会，日本ソフトウェア科学会，IEEE-CS，ACM各会員．



大森　隆行（学生会員）

2003年立命館大学理工学部情報学科卒業．2005年同大学院理工学研究科博士前期課程修了．現在，同研究科博士後期課程在学中．ソフトウェア開発環境の研究に従事．