ソースコード編集履歴のリファクタリング手法

A Technique for Refactoring Editing Histories of Source Code

林 晋平*　大森 隆行†　善明 晃由‡　丸山 勝久§　佐伯 元司¶

あらまし　本稿では，ソースコード編集履歴のリファクタリング手法を提案する．ソフトウェア開発では，ソースコード本体のみならず，その編集の履歴もさまざまに利用されるため，適切に管理された履歴が後段のプロセスに貢献する．しかし，現実には複雑に絡み合った不適切な履歴が生じうる．我々のアプローチでは，編集履歴を，理解性や利用性を向上させるために，その編集内容を変えないよう書き換えるリファクタリングを行う．我々は 4 つの基本的なリファクタリング及びそれらを組み合わせた 2 つの大きなリファクタリングを，それらの事前条件も含めて定義した．また，定義したリファクタリングを自動化しコードエディタに組み込んだ支援ツールを実現した．支援ツールを用いることにより，実際に編集履歴のリファクタリングが行えること，またリファクタリングにより分割コミットや変更取り消しなどの有益な応用が可能となることを示す．

1　はじめに

ソースコード本体のみならず，その編集の記録（**編集履歴**）もソフトウェア開発のなかで利用されており，開発の効率化や理解に役立っている．例えば，開発者はコードエディタが保持している編集履歴を利用し，自身がこれまでに行った編集操作を取り消す Undo 操作や，Undo した操作を再度適用し直す Redo 操作を駆使し，さまざまな試行錯誤を行ったり，自身の行った編集内容を確認したりしている．また，版管理リポジトリが蓄えるソースコードの改版の情報も，開発者が行った一連の編集結果に基づくものであり，編集履歴を反映している．さらに，改版履歴のみならず，開発者によって行われた編集履歴をそのまま蓄え，過去の編集操作を再生することにより，ソフトウェア理解を助ける試みも行われている [8] [13] [14]．

以上のように開発者によって行われた編集履歴の利用範囲は広いものの，その有益な利用には課題が残る．例えば，ソフトウェア構成管理において，単一のタスクに関連する小さな変更ごとにコミットすべきとするポリシー（Task Level Commit）[4] が広く知られており，編集結果の差分がこのポリシーを満たしていれば好ましい．しかし，実際のソフトウェア開発では，リファクタリングや，振る舞いに影響しない些細な変更（Non-essential change）が通常の機能追加などの変更と入り交じって行われていることが報告されている [10] [12]．このことからわかるように，複数のタスクに関連する変更がワーキングコピー上で混在するという事態は多く起こり，その結果得られる差分は Task Level Commit に適したものとはならず，修正を必要とする．同様に，複数のタスクに関連する編集操作が入り交じった編集履歴から，特定の種類の編集のみを Undo したりすることも既存のエディタでは難しい．Eclipse をはじめとする開発環境の発展により，例えばリファクタリング操作がダイアログを経由せず通常のコード編集とシームレスに行えるなど，開発者は多様な変更を容易に組み合わせることが可能となったが，このことは履歴の複雑化を助長する可能性がある．以上のことは，理解や利用のしやすい履歴の構造と，ソフトウェア開発

*Shinpei Hayashi, 東京工業大学
†Takayuki Omori, 立命館大学
‡Teruyoshi Zenmyo, 東京工業大学
§Katsuhisa Maruyama, 立命館大学
¶Motoshi Saeki, 東京工業大学

において頻発する履歴の構造の間にギャップがあることを示しており，行った変更をそのまま記録せず，理解や利用に適した形になるよう整理することの必要性を示唆している．さらに，例えば外部サービスと接続するプログラムの実現時に不用意にパスワードをソースコードに記述してしまった場合には，編集履歴をそのまま永続化することは好ましくなく，その一部を取り消せることが望ましいなど，不適切な情報を削除するという観点から履歴の書き換えが望まれる場合もある．

編集履歴の修正は，履歴の再適用時に矛盾が発生しないように行うことが望ましい．編集操作はソースコード上の字句操作に対応しているため，ソースコードの他の箇所を書き換えている編集操作の編集位置に影響を及ぼす．このことは，一見些細にみえる履歴の修正要求に対しても，実際には履歴全体にまたがる修正が必要となることを示しており，履歴の修正手法の自動化が望まれる．

本稿では，**ソースコード編集履歴のリファクタリング**手法を提案し，またその自動化ツールを示す．これは，当然ながらソースコードのリファクタリング [7] から発想した概念で，コードエディタが保持する過去の編集操作の履歴を，その効果を変えないまま利用や理解が容易になるよう書き換えるものである．我々は，4 つの基本的なリファクタリングと，これらを組み合わせて構成した 2 つの大きなリファクタリングを具体的に定義した．また，定義したリファクタリングを自動化する履歴リファクタリングブラウザを Eclipse 上に実現した．ツールは現在のところプロトタイプに留まるが，これを用いることにより，コードエディタにより発行された編集履歴に対して，定義したリファクタリング操作を自由に適用することができている．また，このツールにより，複数の意図に属する一連の変更が混在した履歴を分割して版管理リポジトリにコミットすることや，一部の変更を差し戻す選択的 Undo の応用が実現できた．

本稿の主要な貢献は，新概念として編集履歴のリファクタリングを提唱したことと，カタログの定義及びツールの実装によりその実現可能性を示した点にある．これらにより，前述した有用な応用例が示せただけでなく，例えば編集履歴をリファクタリングすべき兆候として「履歴の臭い（bad smells in editing history）」を考えることができる等，既存のリファクタリングに関する概念を転用することにより，将来に向けて解決すべき課題も明らかとなる．履歴書き換えの試みはこれまでにも存在するものの（詳細は 4 節を参照），提案手法は既存のリファクタリングの概念になぞらえて書き換えを定義した点，改版履歴ではなく編集履歴を対象として，開発環境と密に連携する応用ツールを作成している点が異なる．

本稿の以降の構成を以下に示す．2 節では我々の提案する履歴リファクタリング手法について述べ，そのカタログを示す．3 節ではリファクタリングの自動化ツールの設計と実装を示し，ツールを用いたソフトウェア変更における応用例も紹介する．本研究に関連する研究を 4 節で紹介し，最後に 5 節で本稿をまとめる．

2　編集履歴のリファクタリングとそのカタログ

本稿では，編集履歴のリファクタリング[1]を提案する．編集履歴のリファクタリングとは，履歴全体が表している変更の効果を変えないまま，その理解や利用が容易となるよう履歴の内容を再構成することを指す．変更の効果を変えないとは，同一の対象に対して履歴中の全変更を適用すれば同一の結果が得られることを意味する．理解のし直しや錯誤を避けるため，すでに共有された変更を再構成すべきではない．よって，履歴リファクタリングは他の開発者と変更内容を共有する前に行う．編集履歴を保守・拡張することは考えがたいため，コードリファクタリングと異なり履歴リファクタリングは拡張性の向上を目的としない．これより，コスト対効果の観

[1]本稿では，ソースコードと編集履歴，適用先の異なる 2 種類のリファクタリングを論じる．以降，文脈から自明な場合を除き，これらを「コードリファクタリング」「履歴リファクタリング」と区別する．

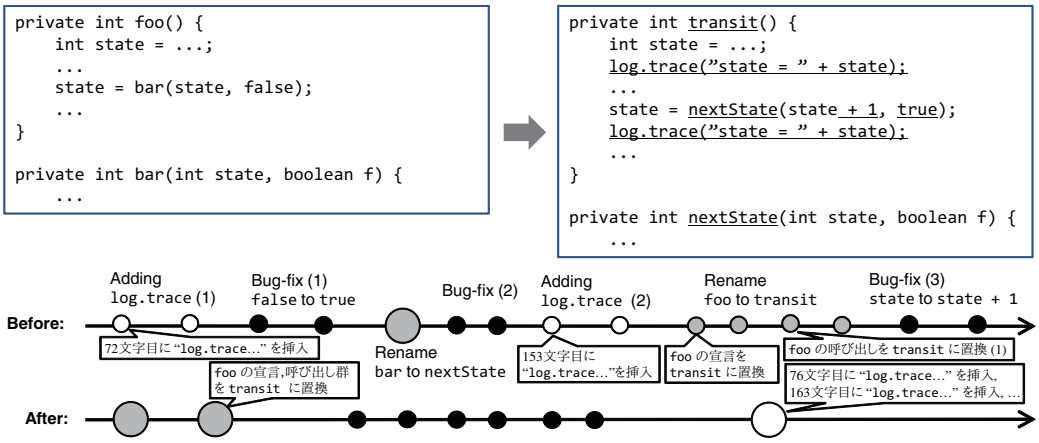


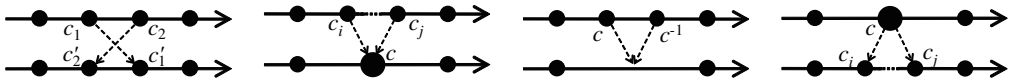
図 1　編集履歴のリファクタリング例

点から，履歴リファクタリングは特に低コストで行える必要がある．また，編集履歴は開発者のプログラミングタスクの最中に蓄えられ，またそのタスクで利用されるため，履歴リファクタリングはプログラミングタスクを阻害しないようシームレスに素早く行える必要がある．これらのことは，履歴リファクタリングの自動化の重要性を示唆している．

図 1 に履歴リファクタリングの例を示す．図左上部のソースコードを編集した結果が右上のソースコードであり，この例ではメソッド `foo` に潜むバグ修正を行う傍らメソッド名の変更も行っている．下線は修正箇所を表す．図下部の実線矢印は時系列を，丸は変更を表し，上部の矢印（Before）が履歴リファクタリング前の編集履歴である．開発者はまず，ロギングコード `log.trace(···)` を挿入し，プログラムを実行しその挙動を確かめている．その後，メソッド `bar` の第二引数を変更している．また，`bar` の名前が不適として，リファクタリングブラウザを用いてこれを `nextState` に変更している（履歴においてこの変更が大きい丸で表されているのは，複数の箇所の編集を含んでいるためである）．その後，他の箇所も修正するがこれはバグを解決するに至らず，最終結果には反映されない．開発者はさらにロギングコードを追加し，またメソッド `foo` の名称を手動で `transit` に変更し，最後に `nextState` の第一引数も変更し，バグ修正を終了している．

この履歴は，開発者の試行錯誤をそのまま反映しているため，様々な意図の変更を含んでおり，理解しづらい．また，`foo` の名前変更は手動で行われたため，識別子の修正が別個に扱われている．さらに，ログ収集コードは，今後の開発に不要と判断されれば削除すべきであるが，履歴中ではその追加変更が複数の箇所に分散しており，変更の取り消しが容易でない．これらを解決すべく，履歴をリファクタリングした結果が下部の矢印（After）である．まず，コードリファクタリング，バグ修正，ロギングコード挿入のそれぞれが連続しており，変更を版管理リポジトリに個別にコミットすることが容易である．また，手動で行ったリファクタリングは単一の変更にまとめられている．さらに，ロギングコード挿入は最近に行われたように位置づけられており，開発者は検討の後，これらが不要であれば容易に変更を Undo できる．履歴リファクタリング前にはソースコードの 72 文字目への挿入と記録されていたログ収集コードの追加は 76 文字目への挿入に改められているなど，履歴全体の整合性を保つよう編集のオフセットが修正されている．

以降，本稿での編集履歴の定義を述べ，具体的なリファクタリング操作のカタログを示す．ここで，履歴リファクタリングは時系列上のデータを扱うことから，その設定も煩雑となり得る．設定を容易とするため，編集履歴は前例での「コードリ



(a) Swap Changes (Swap)　(b) Merge Changes (Merge)　(b′) Cancel Changes (Cancel)　(c) Split Change (Split)

図 2　基本的な履歴リファクタリング

ファクタリング」や「バグ修正」などの，変更のグルーピング（図 1 での変更の色）を表現できるようにし，グループに基づいて履歴リファクタリングを設定可能とする．なお，本稿では編集履歴を対象としてリファクタリングを定義しているが，この定義はそのまま改版履歴を対象としても適用可能であると考える．

2.1　定義

本稿では，特定のソースコードに対する連続する字句の追加や削除，あるいは置換を**編集片**（hunk）と呼び，組 $h = (t, f, o, r, a)$ で表す．ここで，t は編集が行われた時刻，f は編集対象のファイル，o は編集の開始オフセット，r は編集前のファイルから削除された文字列，a は編集後のファイルに追加された文字列を表す．h が追加を表す場合の r，削除を表す場合の a は空文字列となる．$\mathrm{len}(h) := a.\mathrm{length} - r.\mathrm{length}$ は編集片におけるソースコード全体の文字長の変化である．

編集履歴 $H := c_1 \cdots c_N$ の各要素は**変更**（change）であり，これは編集片の順序列と変更が属しているグループ $g \in G$ の対 $c = (h_1 h_2 \cdots h_n, g)$ として表す．各編集片や変更の大きさ，所属グループは $c[i] \equiv h_i, |c| := n, \mathrm{g}(c) := g$ として参照できる．

開発者が端末に対して行う様々な操作に対応して変更が構築される．例えば，通常のタイピングによるソースコード記述では，（主に追加の）単一の編集片を持つ変更が構築され続ける．一方，エディタの置換コマンドによる一括置換，IDE のリファクタリングブラウザが備える各種のコードリファクタリング操作等からは，様々な箇所への置換の編集片が複数連なった変更が構築される．

編集履歴のリファクタリングは，編集履歴の書き換えであり，（1）編集履歴が含む一連の変更全体に対して効果を変えないよう，また（2）リファクタリング前後での全編集片に 1 対 1 対応が構成できるよう，変更を追加・削除したり，各変更の構成を修正する．条件（1）は，同一のソースコード群 S に対してリファクタリング前後の編集履歴中の変更を順に適用すれば，等しいソースコード群 S' が得られることを述べている．条件（2）は，提案するリファクタリングが編集片を基本単位としてその内部を変更せず，その構成に注目していることを指しており，開発者がひとつの塊として行ったソースコード編集を複数に分断することを防いでいる．

当然ながら，履歴リファクタリングの結果得られた編集履歴は，開発者が本来行った変更の記録とは異なるため，目的によっては適さない場合がある．例えば，履歴に残った開発者の行動から，開発者の能力を評価する [16] 等の目的には適さない．こういった目的のためには，リファクタリング前の元々の編集履歴も必要に応じて保存しておいてよい．

2.2　基本的なリファクタリング

我々は，4 つの基本的なリファクタリングを定義した．これらのリファクタリングの概要を図 2 に示す．図中の実線矢印は時系列を，丸は変更を表し，上部をリファクタリングしたものが下部である．変更を行う順番を入れ替える Swap，複数にまたがる変更を単一として構成する Merge，逆の意味の変更を打ち消して削除する Cancel，変更を特定の観点（例えばクラスやメソッド毎）から複数に切り分ける Split がある．Cancel は，Merge の特殊な場合として，マージした結果の変更群が効果を及ぼさない場合に，変更そのものも削除してしまう操作である．

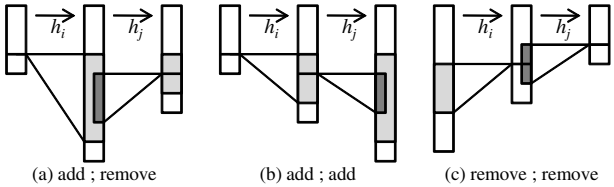


図 3　編集片の交換の失敗例

コードリファクタリングと同様，履歴リファクタリングにも事前・事後条件を定義できる．例えば Swap の事前条件は，簡単に言えば後段の変更 c_2 が前段の変更 c_1 に依存していないことである．また，いずれのリファクタリングも，変更の効果が不変であることが事後条件として定義できる．多くの場合，この事後条件は容易に確かめられる．

各リファクタリングの定義は以下の通りである．

名前　Swap($H \equiv c_1 \cdots c_N, c_i, c_j$)
入力　編集履歴 H，H 中での交換対象 c_i, c_j（両者は H 中で隣り合う：$i+1=j$）．
操作　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($c'_i[k], c'_j[l]$).
　　3. $H \leftarrow c_1 \cdots c_{i-1} c'_j c'_i c_{j+1} \cdots c_N$.

名前　Merge($H \equiv c_1 \cdots c_N, H' \equiv c_i c_{i+1} \cdots c_j$)
入力　編集履歴 H，結合対象の変更群 H'（H' 中の変更は H 中で隣り合う）．
事前条件　全対象が同一グループに属している．$\mathrm{g}(c_i) = \mathrm{g}(c_k)$ for all $i < k \leq j$.
操作　1. $c' \leftarrow (c_i[1] \cdots c_i[|c_i|] c_{i+1}[1] \cdots c_{i+1}[|c_{i+1}|] \cdots c_j[1] \cdots c_j[|c_j|], g)$.
　　2. $H \leftarrow c_1 \cdots c_{i-1} c' c_{j+1} \cdots c_N$.

名前　Split($H \equiv c_1 \cdots c_N, c_i, B \equiv b_1 \cdots b_n$)
入力　編集履歴 H，H 中の分割対象 c_i，分割境界 $1 < b_1 < \cdots < b_n \leq |c_i|$.
操作　1. $c'_0 \leftarrow (c_i[1] \cdots c_i[b_1 - 1], \mathrm{g}(c_i)), \ldots, c'_n \leftarrow (c_i[b_n] \cdots c_i[|c_i|], \mathrm{g}(c_i))$.
　　2. $H \leftarrow c_1 \cdots c_{i-1} c'_0 \cdots c'_n c_{i+1} \cdots c_N$.
補足　分割境界は，各編集片が編集している構文要素の違い等により定める．注目する構文要素間をまたがる編集片が存在する場合，Split は失敗する．

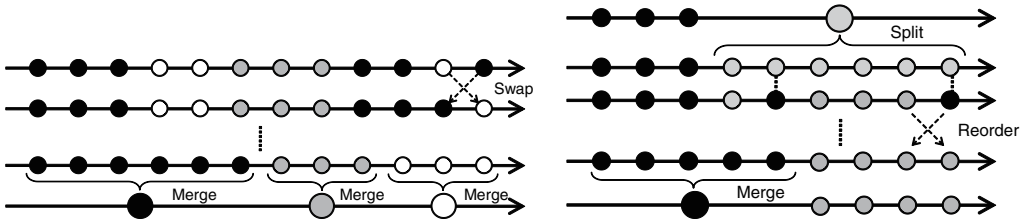
Swap は，編集片の交換 Commute(h_i, h_j) に基づき定義されている．これは，編集片 $h_i = (t_i, f_i, o_i, r_i, a_i)$ と $h_j = (t_j, f_j, o_j, r_j, a_j)$ がこの順で行われた想定で，これらを逆順で行われたよう編集開始オフセットを書き換えるもので，

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

と行う．ただし，異なるファイルにおける編集片（$f_i \neq f_j$）の場合，なにも行わない． 編集片間に依存関係がある場合，交換できない．交換が失敗する場合は，例えば図 3 に示す場合であり，これらはそれぞれ（a）追加した領域の文字列を削除する場合,（b）追加した領域の内側に別の文字列を挿入する場合,（c）削除した領域を覆う領域を削除する場合である．このように，編集片を交換した場合に，削除対象が存在しなくなったり，編集領域が複数の領域に分断される場合には，履歴リファクタリングの条件（2）を満たせなくなるため，交換を失敗させる．

2.3　大きなリファクタリング

複数のリファクタリングを合成することにより，**大きなリファクタリング**を定義できる．本カタログで定義した大きなリファクタリングの概要を図 4 に示す．Reorder は，さまざまなグループに属する変更が混在している履歴を，グループ毎にまとめ



(a) Reorder [and Merge] Changes (Reorder)　(b) Reconfigure Changes (Reconfigure)

図 4　大きなリファクタリング

る [9]．Reconfigure は，着目する大きな変更のうち，それまでに行った変更に関連のある部分のみ取り出してまとめるもので，変更を分割した後，必要な部分を集め，マージするという一連のリファクタリングから成っている．

Reorder では，まず Swap を繰り返し適用し，同種のグループの変更を近接させる．Swap を適用する基準として，グループ間の順序関係を与える．次に，必要に応じて Merge により，各グループの変更群を一つにまとめる．

名前　Reorder($H \equiv c_1 \cdots c_N, H' \equiv c_i \cdots c_j, \preceq \subseteq G \times G$)
入力　編集履歴 H，H 中の部分列 H'，グループ間の順序関係 $\preceq$.
操作
1. $\mathrm{g}(c_i) = \mathrm{g}(c_k) \Longrightarrow \mathrm{g}(c_i) = \mathrm{g}(c_j)$ for all $i < j < k$ が成り立つよう，$\preceq$ に基づき H 中の部分列 H' をバブルソートする．ソートにおける要素の交換時には Swap を適用する．
2. 必要に応じて，同一グループに属する隣り合う変更をマージしまとめる．Merge($H, c_i \cdots c_j$) for all $i \cdots j$ $s.t.$ $\mathrm{g}(c_i) = \cdots = \mathrm{g}(c_j)$.

当然ながら，順序関係 $\preceq$ の取り方によっては Swap が失敗してしまう．グループ順序が従うべき事前条件の詳細は，文献 [9] の手法に基づく．バブルソートは隣り合う要素のみを交換するため，Swap を併用する目的に適している．

Reconfigure は，例えば名前変更のコードリファクタリングや空白文字のフォーマットなど，ソースコードやプロジェクト全体にまたがって行われる大きな変更を，それまでに行った変更に関係する部分のみに局所化して扱いたい場合に用いる．まず，着目する変更を Split により複数の変更に分割する．分割には，クラス毎やメソッド毎など，構文要素の粒度を用いる．次に，既存の変更が関わっている構文要素を対象とした変更群を，新しいグループに所属させる．さらに，Reorder を適用することにより，これらの変更群を過去の変更群に近接させる．最後に Merge により，着目するグループの変更群を一つにまとめる．

名前　Reconfigure($H \equiv c_1 \cdots c_N, H' \equiv c_i \cdots c_j, c$)
入力　編集履歴 H，着目する変更群 H'，分割対象の変更 c.
操作
1. Split(H, c, B)：着目する観点（メソッド）で，対象の大きな変更を分割し，$c'_1 \cdots c'_n$ を得る．
2. Reorder($H, c'_1 \cdots c'_n, \preceq$)：$\mathrm{g}(c_j)$ を最左とするグループ順序 $\preceq$ を用いて 1 の結果を並べ替える．
3. Merge($H, H' c'_1 \cdots c'_i$)：着目する変更群と同一グループに属する隣り合う変更をマージし，まとめる．

3　ツールによる自動化

3.1　概要

我々は，履歴リファクタリングを自動化する支援ツールのプロトタイプ Historef を実現した．Murphy-Hill が述べているように，リファクタリングツールの利用プロセスにはリファクタリングの実行だけでなく，その設定（configure）や取り消し（undo）などが含まれる [11]．編集履歴のリファクタリングツールにおいても，これ

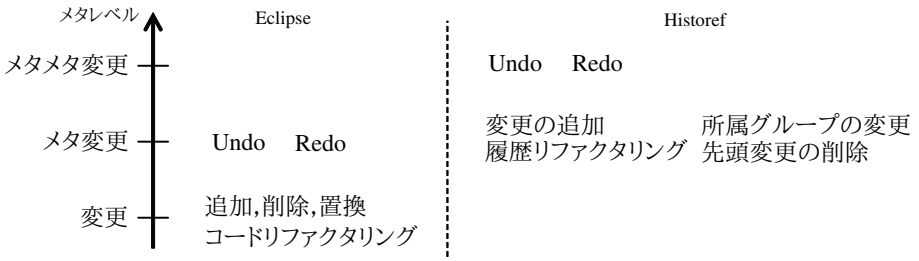


図 5　Historef における変更の扱い

らの支援が望まれる．Historef に対して課した要件は以下の通りである．

- 履歴に対して 2 節で示したリファクタリングが実行できること．
- 履歴リファクタリングの設定に対する支援があること．
- 履歴リファクタリングの Undo・Redo が行えること．

Historef は Eclipse プラグインとして構成されており，編集履歴取得ツール OperationRecorder [18] と連携している．ツールは大きく，履歴管理系とリファクタリングブラウザの 2 部から構成されている．管理系は，OperationRecorder から得た編集履歴を分類し，開発者に対して表示する部分までを含む．リファクタリングブラウザでは，前節で述べた主な履歴リファクタリングを自動化しており，開発者が自由に履歴リファクタリングを実行できる．

Historef は，従来のエディタが保持する編集操作の履歴バッファとはレベルの異なる履歴を保持している．Historef における変更の扱いの概要を図 5 に示す．Eclipse 本体においては，変更（Operation）群が取り消しバッファに蓄えられ，またこれらが Undo・Redo の対象となる．一方，Historef では変更レベルの履歴は単なるデータとして扱い，変更を書き換えるメタ変更の列を Undo・Redo を行うためのコマンド列として保持する．OperationRecorder から変更を取得した際，Historef は「該当変更が履歴に追加された」というメタ変更を構成し，これをメタ変更履歴に追加する．履歴を書き換える履歴リファクタリングも同じくメタ変更として定義され，実行時には同様にメタ変更履歴に追加される．従来のエディタで変更の Undo・Redo が行えるのと同様に，Historef ではメタ変更履歴における Undo・Redo が定義されており，開発者に試行錯誤の余地が与えられる．変更を追加するメタ変更を Undo・Redo することにより，変更レベルでの Undo・Redo が実現されるため，メタ変更としての Undo・Redo を特別なコマンドとして用意する必要はない．ただし，後述する応用例 2 での選択的 Undo を実現するため，メタ変更履歴の先頭要素を削除するための特別なコマンドのみを追加している．

3.2　動作例

Historef の動作例を図 6 に示す．この例では，開発者が Historef を起動した後に，ソースコードにコメントの追加や変数名変更のコードリファクタリングを行ったあとを示している．画面右部の Changes ビューには，行われた変更が並んでおり，その実行時刻や変更の内容が俯瞰できる．

Historef は，リファクタリングの設定を支援するための履歴のグルーピング機能を持っている．この開発環境では，編集のモードを定めることのできる特別なコードエディタ [9] を用いてソースコードの編集を行う．例えば，モード 1 で編集した際に得られた変更はグループ 1 に属する．開発者は，ショートカットキーを用いて自由にモードを切り替えながらコードを編集することにより，変更を容易にグルーピングできる．また，すでに行われた過去の変更の所属するグループを修正することもできる．図 6 の例では，(1) バグ修正と（2）コードリファクタリング，(3) コメントの追加の 3 グループが定義されており，それぞれに変更が分類されている．変

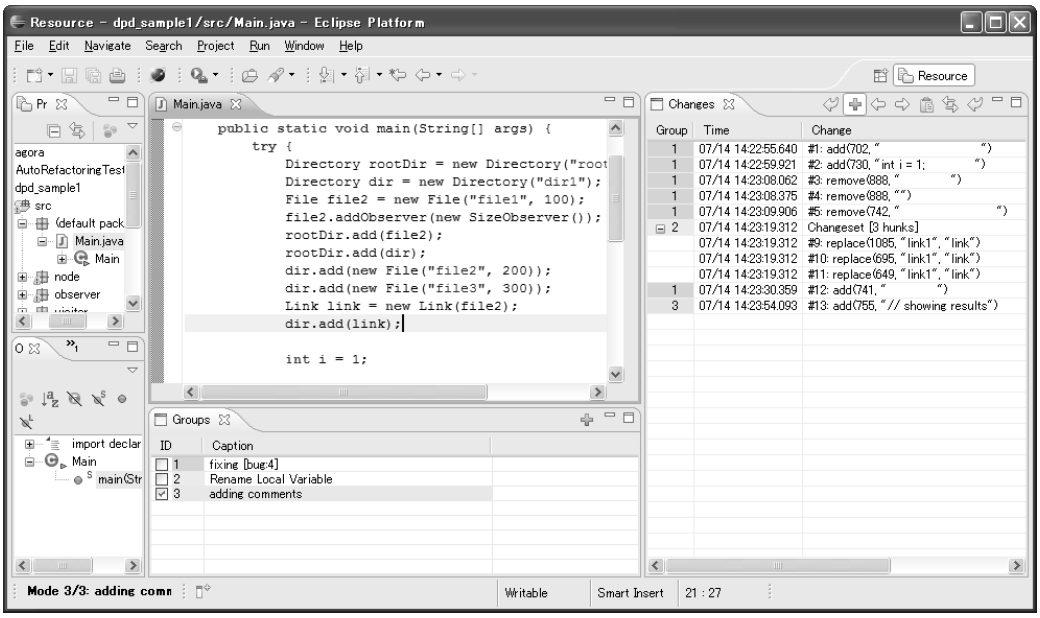
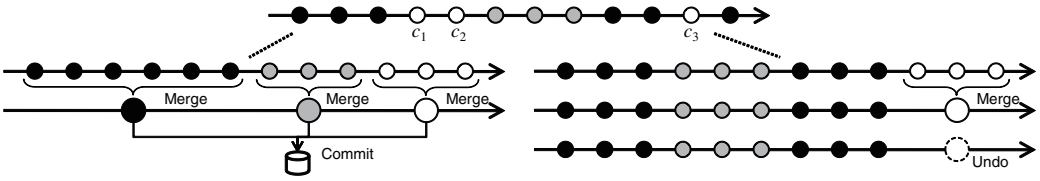


図 6　Historef の動作例



(a) 分割コミット　　(b) 選択的 Undo

図 7　履歴リファクタリングの応用例

更のグループは Groups ビューに一覧表示されている．コードエディタ上で行った変更は，Groups ビューで現在選択されているグループに自動的に属する．各グループにはキャプションを入力することができ，ここで入力したキャプションは後述する分割コミットの際のコミットログの初期値として用いることができる．

グルーピングの手間を省くために，一部の変更に対しては自動的に新しいグループを割り振ることもできる．例えば，グループ 2 のコードリファクタリングは，Eclipse のコードリファクタリング機能を用いて実行されたもので，これは Eclipse が発行した変更に付与された名前をそのまま利用し，新しいグループに属する変更として自動的に登録させている．

3.3　応用例

Historef の履歴リファクタリングを用いることにより，以下の応用を実現できている．

3.3.1　分割コミットへの利用

Task Level Commit を実現するため，Reorder リファクタリングによりグループ毎に並べ替えられ，マージされた各変更をそれぞれリビジョンとして版管理リポジトリにコミットすることができる（図 7(a)）．分割コミットは，マージ後の各変更をパッチとみなし，全編集履歴の適用前のソースコード群に対して各パッチを順に行った際のそれぞれの状態をコミットすることで行う．

3.3.2 変更の選択的 Undo

直近に行われた変更のみならず，編集履歴の任意の変更（群）を取り消す，選択的 Undo [5] も実現されている．選択的 Undo の概要を図 7(b) に示す．選択的 Undo も，編集履歴の並べ替えに基づいており，まず着目する取り消し対象の変更群（この例では c_1, c_2, c_3）を Swap により最近に移動させる．次に，移動した変更群を Merge によりひとつにまとめる．最後に，マージされた変更に対して Undo 操作をかけ，現在のソースコード群から変更の影響を除去する．Historef には，この一連の操作をひとまとめに行う選択的 Undo コマンドが実現されており，開発者は（関連する履歴リファクタリングが成功する場合に限るが）過去の変更を自由に取り消せる．

3.4 制限

現在の Historef には，以下の制限がある．

- 本稿での編集履歴のモデルはファイルの追加や削除といったプロジェクトの編集操作を扱えず，これらは履歴リファクタリングや取り消しの対象外となる．
- Historef は Eclipse 本体の取り消しバッファを用いず独自のものを用意しており，履歴リファクタリングや選択的取り消しはこの独自のバッファを対象に行われる．よって，Eclipse 本来の取り消しバッファを利用した（プラグイン等の）機能とは協調動作しがたい．

4 関連研究

ChEOPS [6] や SpyWare [15] をはじめ，理解や再利用のための変更モデル及びそれに基づく開発環境が提案されている．しかし，これらが用意しているツールは，主に構造エディタによる変更の発行に基づいており，テキストエディタ上で試行錯誤を行う現実の開発者のアクティビティと合致していない．

例えば Git [2] や Darcs [1] などの既存の版管理システムでも行われているように，変更履歴を修正する試みそのものは目新しくない．しかし，これらの手法では履歴書き換えをリファクタリングという形では示していないため，事前条件や合成などの概念が系統立てて示されておらず，またそれらを支援するツールも実現されていない．また，これらの手法は主に改版履歴を対象としており，編集履歴に着目して IDE と継ぎ目なく連携するツールを構築しておらず，本稿の応用例で示したような編集作業を支援しない．Mercurial [3] は，MQ と呼ばれる変更（パッチ）のスタックを保持することにより，ワーキングコピー上で複数の変更のグループを管理することを可能としている．しかし，これにより管理される変更は操作ではなく版に基づくものであり，本研究で扱っているような編集操作間の依存関係を扱っていない．

本稿での研究成果は，筆者らの過去の成果 [9] を発展させたものである．この論文では編集操作の並べ替え手法を提案しているが，これは分割コミットの支援に特化したもので，リファクタリングとして定義されたものではなく，また統合された自動化ツールも実現できていなかった．本稿での成果はこれに比べて改善されている．

5 おわりに

本稿では，編集履歴のリファクタリング手法を提案した．また，編集履歴を対象に提案手法を実現するツールを作成した．ツールの使用性等の評価は行えていないものの，提案した履歴リファクタリングは実現可能であり，また 2 つの有用な応用が実現できることがわかった．

編集履歴の書き換えをリファクタリングとして表したことの利点のひとつに，リファクタリングに関わる多くの概念を借用できることがある．基本的なリファクタリングを組み合わせて大きなリファクタリングを構成した前例はそのひとつであるし，編集履歴の不適切さを判断するための尺度を「履歴の臭い」として計測することにより，版管理リポジトリにコミットを行う前に変更の再構成を促せる可能性が

ある．例えば，前述した，特定の意図（グループ）の編集が履歴全体に分散している場合は，編集履歴における変更の分散（Shotgun Surgery）という臭いと考えることができる．その他にも，臭いの候補としては例えば差分レビューのために用いる理解性のメトリクス [17] の利用が考えられる．

今後の課題として以下を考えている．

- **履歴の臭いの定義**．前述したものをはじめとして，臭いのカタログを整備し，履歴リファクタリングとの関係性を明らかにする．
- **履歴リファクタリング手法の評価**．履歴リファクタリングは現在のところ自動化までに留まっており，ユーザビリティや有用性等の評価はまだ行えていない．
- **変更モデルの拡張**．本稿で示したモデルは，版管理におけるブランチのような枝分かれを扱えない．また，大きなコードリファクタリングが小さなコードリファクタリングの集まりとして構成されているといった，変更の親子関係も扱えない．これらを扱えるよう編集履歴のモデルを拡張し，ツールの支援を拡充させることが有用と考える．

謝辞 本研究の一部は科研費（21500043，23700030）の助成を受けたものである．

参考文献

[1] Darcs. available at `http://darcs.net/`.
[2] Git. available at `http://git-scm.com/`.
[3] Mercurial scm. available at `http://mercurial.selenic.com/`.
[4] S. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.
[5] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.
[6] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proc. 2007 International Conference on Dynamic Languages* (*ICDL 2007*), pp. 3–24, 2007.
[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[8] L. Hattori, M. Lungu, and M. Lanza. Replaying past changes in multi-developer projects. In *Proc. 4th International Joint ERCIM/IWPSE Symposium on Software Evolution* (*IWPSE-EVOL 2010*), pp. 13–22, 2010.
[9] S. Hayashi and M. Saeki. Recording finer-grained software evolution with IDE: An annotation-based approach. In *Proc. 4th International Joint ERCIM/IWPSE Symposium on Software Evolution* (*IWPSE-EVOL 2010*), pp. 8–12, 2010.
[10] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proc. 33rd International Conference on Software Engineering* (*ICSE 2011*), pp. 351–360, 2011.
[11] E. Murphy-Hill. A model of refactoring tool use. In *Proc. 3rd Workshop on Refactoring Tools* (*WRT 2009*), 2009.
[12] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proc. 31st International Conference on Software Engineering* (*ICSE 2009*), pp. 287–297, 2009.
[13] T. Omori and K. Maruyama. An editing-operation replayer with highlights supporting investigation of program modifications. In *Proc. 2011 12th International Workshop on Principles on Software Evolution and 7th ERCIM Workshop on Software Evolution* (*IWPSE-EVOL 2011*), pp. 101–105, 2011.
[14] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, 2007.
[15] R. Robbes and M. Lanza. Towards change-aware development tools. Technical Report 2007/06, University of Lugano, 2007.
[16] Y. Takada, K. Matsumoto, and K. Torii. A programmer performance measure based on programmer state transitions in testing and debugging process. In *Proc. 16th International Conference on Software Engineering* (*ICSE 1994*), pp. 123–132, 1994.
[17] 保田, 森崎, 松本. ソースコードの差分情報を用いたコードレビューコストの分析. 情報処理学会研究報告, 2009-SE-163(25):193–199, 2009.
[18] 大森, 丸山. 開発者による編集操作に基づくソースコード変更抽出. 情報処理学会論文誌, 49(7):2349–2359, 2008.