編集操作履歴の階層的なグループ化を用いたポリシー準拠のコミットの構成支援

松田 淳平[1]　林 晋平[1]　佐伯 元司[1]

概要：版管理システムを用いたソフトウェア開発において， コミットポリシーに従ったコミットは開発者らにとって有益である．しかしポリシーに従ったコミットの構成は開発者の負担となるため，その支援が望まれる．提案手法では，ポリシーに従った粒度のコミットをソースコード編集後に得るために，開発者による編集操作履歴を，開発環境が提供する編集の種類を用いて階層的に管理する．開発者が階層の構成要素である節を指定することにより，対応する粒度のコミットの列を，変更全体の整合性を維持したまま構成することができる．手動編集を含む大きなリファクタリングを適用したソースコードと記録した編集操作履歴に対し，本手法を適用した結果，複数のポリシーが定める粒度のコミットをそれぞれ適切に構成することができた．

1. はじめに

版管理システムを用いたソフトウェア開発において， コミットポリシーに従ったコミットは開発者らにとって有益である．開発者が他の開発者らのコミットを理解することは，編集差分の理解や変更の再利用に役立つ [5]．そのためプロジェクトや組織などによってコミット 1 つの大きさやコミットの含む意味を記述したポリシーが規定されている．例えば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 Task Level Commit [3] は広く知られているポリシーの 1 つである．しかしながら一般に，開発者は様々な意味が混ざり合った状態の変更を行い，コミットを行ってしまうことがある [6], [7], [9], [10]．ポリシーに従うコミットを構成する際，混ざり合った変更を行わないよう，開発者は行う編集の意味や順序を考慮し，コミットのタイミングを見極める必要があるため，開発者の負担となる．版管理システムにはすでに行われたコミットを統合したり，分割したりするなどの機能が備わっているものもあるが，その実行のためには対象となるコミットの深い理解が必要となる．また，これらの機能は，編集から適切なコミットの構成を支援する目的で用意されたものではなく，依然として開発者に対する負担が残るという問題がある．

そこで提案手法ではポリシーが期待する粒度のコミットをソースコード編集後に得るため，開発者による編集操作履歴を，開発環境が提供する編集の種類を用いて階層的に管理する．開発者が階層の構成要素である節を指定することにより，指定された節が表現している粒度のコミットの列を，変更全体の整合性を維持したまま構成する．特に，統合開発環境と連携することにより，リファクタリングや些細な変更 [7] といった変更を，変更の種類と関連付けた形で記録することができるため，これを用いて階層を構成する節を自動的に記録することが可能となる．提案手法を統合開発環境 Eclipse [1] 上に，Eclipse の提供するリファクタリングとそれらの意味的な関係を利用する形で実装した．手動編集を含む大きなリファクタリングを適用したソースコードとその記録された編集操作履歴に対して提案手法を適用し，提案手法の有用性を評価した．

本稿の構成を以下に示す．まず 2 章でコミットの有用性と既存手法では解決困難な問題点を挙げる．3 章では提案手法のアプローチを述べ，4 章では提案手法を実装したツールの説明を行う．5 章で提案手法の適用実験を行い，手法の有用性を示す．6 章では関連研究を挙げ，7 章では本稿をまとめ，今後の課題を述べる．

2. 背景と問題点

コミットとはソフトウェア成果物に対する変更を版管理リポジトリへ保存することであり，また保存された変更そのものを指すこともある．

一般にコミットにはその意味や目的を表現したメッセージ（コミットメッセージ）が付加される． 開発者は行われたコミットの内容やコミットメッセージを読むことによ

[1] 東京工業大学 大学院情報理工学研究科 計算工学専攻
Department of Computer Science, Tokyo Institute of Technology

© 2014 Information Processing Society of Japan

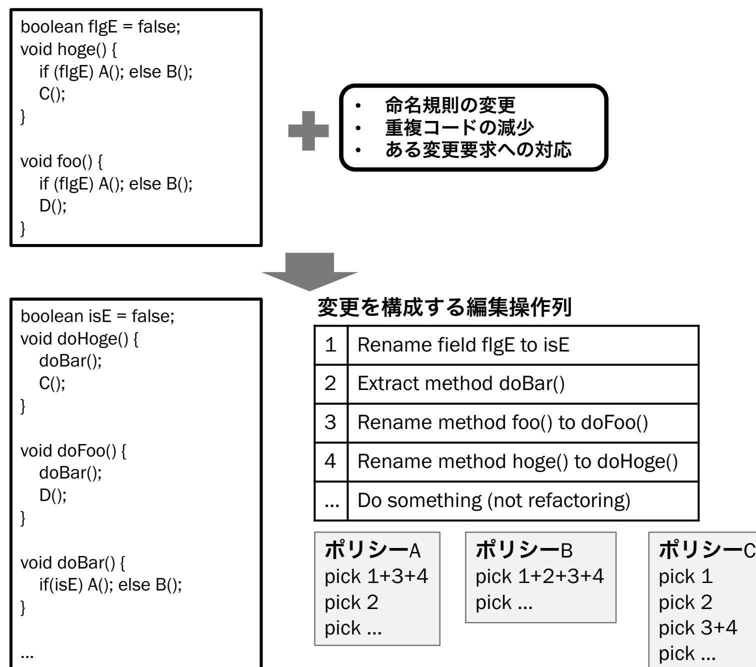

図 1 混在する編集操作とポリシーによる分類

り，他の開発者が行った変更を把握する．他の開発者の変更を理解することは，特に変更レビューにおける編集差分の理解，変更の再利用やコンフリクト発生時の対処の場合に必要となる．そのため，コミットの理解の容易さなどの観点から，コミットポリシーが定義され，開発者はコミットポリシーが期待する粒度でのコミット（ポリシーに従ったコミット）を行う必要がある．広く知られているコミットポリシーとして Task Level Commit [3] が存在し，それは単一のタスクに関連する変更ごとにコミットすべきというものである．しかしながら実際の開発現場では複数の意味が混ざり合った状態でコミットをしてしまうこと [6], [7], [9], [10] があり，複数の意味の混在は編集差分の理解や変更の再利用の妨げとなってしまう．

意味の混ざり合った変更が発生する要因の 1 つとして，意味ごとにコミットを構成することが，開発者にとって負担であることがある．コミットの意味や大きさはコミットの実行時に確定され，それらは最新リビジョンから現在までの編集内容によって決定される．したがって意味ごとにコミットを構成するためには，開発中に同じ意味の編集をまとめるよう，編集順序の考慮を行う必要があり，それは開発者にとって負担となる問題点が存在する．版管理によっては編集順序の考慮をせずに済むよう，開発者が最新リビジョンから現在までの編集内容の中から任意の編集を選択してコミットを行うことが可能である．しかしながら開発者が編集内容全体から同じ意味による編集のみを選ぶことやビルドエラーの有無を判断することは難しく，開発者の負担を回避できていない．

一連の編集後に，そのままではポリシーが期待する粒度のコミットにならない場合，ポリシーに従うよう意味ごとに変更を構成し直す（分割コミット）必要がある．分割コミットの例を図 **1** に示す．図 1 では，ソースコードに対して命名規則の変更，重複コードの集約，ある変更要求への対応の 3 種類の意図からなる変更を行い，編集後のソースコードとその変更を構成した編集操作の列を得ている．命名規則の変更では，メソッド名に do を付与する，フラグフィールドの接頭辞を is へと修正するために，名前変更リファクタリングを実行している．また，重複コードである if 文が別のメソッドとして抽出されている．変更要求への対応は図 1 中左下の枠内に ··· として省略された部分であり，リファクタリングを含まない．このような変更は，変更要求への対応時に floss リファクタリングを実行した際に起こるものであり，また現実の開発でも多発している [9].

開発対象のプロジェクトが従ってるコミットポリシーにより，これらの変更をどのようにコミットすべきであったかは異なってくる．例えば，コミットメッセージに変更の意図としてリファクタリング操作の種類を記述できるよう，異なる意図に基づいて行われた異なる種類のリファクタリング操作は個別にコミットすべきとするポリシーを考えることができる（ポリシー A）．これに対して，プログラムの挙動を変更する通常の変更と，変更しないリファクタリングのような変更を混在させないことに視点を置き，ポリシー A よりも大きなコミットを許容するものもある（ポリシー B）．一方，ポリシー A よりもさらに差分理解を容易にするため，複数種類のリファクタリング操作の混在を制限する考え方もある（ポリシー C）．ポリシー C は，意図による分類に加え，特定の変更意図に基づく複数のリ

© 2014 Information Processing Society of Japan

ファクタリング操作も個別にコミットするもので，いわゆる impure リファクタリング [4] のコミットを許容しない．こういったポリシーは，コミット数の増大をまねくものの，個別のコミットの理解や正しさの検証を容易とする．ここで，今回の変更に対する実際の編集操作列が図 1 中の表で示したように得られたとする．図 1 右下のポリシーはそれぞれ pick の数だけコミットを実行し，pick 右の数字が編集操作列の番号に対応，また + は対応する編集操作による編集を結合させることを意味する．ポリシー A では $\{1, 3, 4\}$（名前変更リファクタリングによる命名規則変更への対応），$\{2\}$（メソッドの抽出リファクタリングによる重複コードの集約），$\{\cdots\}$（変更要求への対応）と変更を分割してコミットすることになる．この分割はポリシー B と C には合致せず，ポリシー B は $\{1, 2, 3, 4\}$ と $\{\cdots\}$ に，ポリシー C では $\{1\}$，$\{2\}$，$\{3, 4\}$，$\{\cdots\}$ と分類を行う必要がある．そのため，図 1 で示した編集全体をそのまま単一の変更としてコミットをしてしまうといずれのポリシーにも従っていない．このように，従うポリシーにより異なる粒度のコミットを構成しなければならない．また本稿では以降，ポリシーが期待するコミットの粒度のことをポリシーの粒度として表現する．

これに対して，編集操作履歴をグループ化することでコミットの分割を図る既存手法がある [14]．しかしながら既存手法のグループ間には特に相互関係がないため，既存手法を用いてポリシーに従ったコミットを構成する場合，細粒度のポリシーに従うためには編集操作履歴に対するグループを細かく大量に用意する必要があり，グループの粒度やその妥当性は開発者自身が判断して行う必要があり，既存手法では前述した問題を解決できていない．また粗粒度のポリシーに従ったコミットを構成しようとする場合，グループの結合を行う必要があるが，グループの結合の順序やその妥当性は開発者自身の判断に頼るところが大きく，やはり既存手法では前述した問題を解決できていない．

このような問題を解決するため，提案する手法では，開発者が開発中に編集順序を考慮することなく，一連の編集後にポリシーに従うようなコミットを容易にかつ柔軟に構成する支援技術の開発を目的とする．

3. 提案手法

3.1 アプローチ

提案手法ではポリシーの粒度に従ってコミットを構成するにあたり，その粒度を適切に選択できるようにするため，変更そのものに構造を持たせ，複数の粒度の変更を表現できるようにする．また構造を維持したまま変更を扱うことにより，粒度の調整を柔軟に行える．さらに表現する粒度と意味を対応づけることにより，コミットポリシーで定義されるような意味のある粒度を表現できる．

そこで図 **2** に示すように，提案手法では開発者による

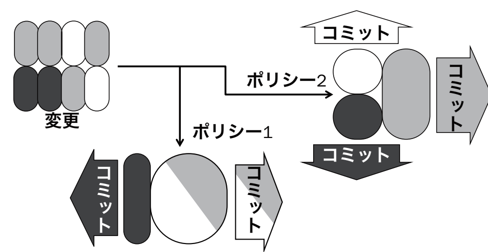

図 **2** 提案手法のアプローチ

ソースコードに対する字句の追加や削除などの編集操作を最小の粒度の変更であるとし，編集操作を収集する．収集した編集操作列が意味ごとの編集操作の順列で表されるよう，編集操作列の順序の並び替えを行う．また変更の意味に基づいて収集した編集操作列を区分するにあたり，節をグループ，枝を意味的な親子関係とした木構造を成す階層性を持ったグループによるグループ化を導入する．根節に近ければ近いほど節が表現する意味の抽象度が上がる構造となる．ポリシーが表現する意味と木構造の節であるグループが表現する意味を対応付けることにより，ポリシーに従った意味ごとに編集履歴を区分することができる．例えば IDE が提供するリファクタリング操作などの変更の意味や種類を利用することで，階層を構成する節の意味付けを行い，対応する 2 つの節の間に意味的な親子関係を構成できる．さらに階層の節から順序関係を抽出することで，提案手法の目的であるコミットポリシーに従う粒度のコミットを構成できるようにする．

3.2 階層を構成する節の定義

提案手法で使用するソースコード編集操作履歴は OperationRecorder [15] によって取得され，Historef [5] によって記録されるため，その編集操作履歴の定義は Historef に準拠する．編集操作履歴 H は変更 c の順序列 $H \coloneqq c_1 c_2 \ldots c_N$ である．変更 c は字句の追加や削除などを表す編集片 h の順序列と，変更が属しているグループ $g \in G$ の対 $c \coloneqq (h_1 h_2 \cdots h_n, g)$ として表す．編集片 h には追加や削除，置換された字句の他に挿入位置の情報を保持している．また複数の変更間には依存関係が存在する場合がある．編集履歴 $H = c_{add} c_{remove}$ が与えられ，変更 c_{add} が文字列 abc を挿入する変更，変更 c_{remove} が変更 c_{add} によって追加された文字列 abc を削除する変更であるとする．変更 c_{remove} は c_{add} によって得られた文字列に対する操作であり，情報を失うことなく 2 つの変更の順序を入れ替えることはできない．このとき変更 c_{remove} は変更 c_{add} に依存しているという．

本手法では既存手法のグループ g に対応する，階層を節の列とし，節 $t \in T$, $t = (id, caption, t_{parent})$ を導入し，変更を $c \coloneqq (h_1 h_2 \cdots h_n, t)$ と再定義する．ここで id は各節に固有の ID，$caption$ は該当節に対応するキャプション文

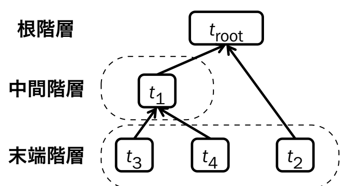

図 3　階層の木構造の例

字列，t_{parent} は t の親の節を表す．また変更 c が属する節 t は $\mathrm{t(c)} \coloneqq \mathrm{t}$ として参照できる．ここで，親を表す節 t_{parent} は，階層構造が木構造となるように，すなわち，（後述する木の根以外で）閉路を持たないように構成する．

前述の定義より，階層集合 T は木構造で表現できる．図 **3** は階層を木構造で表現した例である．節 $t_{root} = (caption, t_{root})$ は木の根となる．変更 c が含むような節 t は木構造における葉に相当するものに限定し，それらの節の集合を $T_{leaves} \subset T$ と表現し，$t \in T_{leaves}$ を末端階層の節と呼ぶ．また節 $t \in T$ が，$t \notin T_{leaves} \cup \{t_{root}\}$ を満たすとき，t は中間階層に属しているという．図 3 中の t_3 及び t_4 は末端階層に属する．すなわち，すべての変更は末端階層に属しており，中間階層は末端階層を構成する複数の節を束ねるための抽象的なカテゴリの役割を持つ．節 t と節 t の子孫に含まれるすべての変更の集合 C_i は $\mathrm{changes(t)} \coloneqq \mathrm{C_i}$ として参照できる．例えば，開発環境の機能を用いたフィールド f の名前変更リファクタリングが末端階層 $t_{Renamef}$ となり，$\mathrm{changes(t_{\mathit{Renamef}})}$ によってその操作によって変化した変数名の置換を表現する変更列が得られる．

また階層的グループ化をされた編集操作履歴を整列させるための順序関係 $\preceq \subseteq T_{leaves} \times T_{leaves}$ は以下の制約を満たす．T_{leaves} の分割 $T' \subseteq 2^{T_{leaves}}$ を考える．ここで T' は $\forall t_1 \neq t_2 \subseteq T' \bullet t_1 \cap t_2 = \emptyset, T' = \bigcup_{t' \in T'} t'$ を満たす．すなわち T' は T_{leaves} の全要素を網羅しており，T' の要素である任意の 2 つの集合間に共通する要素は存在しない．いま順序関係 $\preceq \subseteq T_{leaves} \times T_{leaves}$ と階層を構成する節の集合 $t' \in T'$ が存在し，任意の 2 つの節 $t_1 \neq t_2 \in T_{leaves}$ を用いて $t' = \{\cdots, t_1, t_2, \cdots\}$ と表されるとき，$t_1 \preceq t \preceq t_2$ または $t_2 \preceq t \preceq t_1$ を満たすような $t \in T_{leaves}$ は存在しない．

3.3　変更の再構成

提案手法ではポリシーに従った粒度のコミットを構成するために，階層を利用した編集操作の並び替えを用いて変更の再構成を行う．図 **4** は変更の再構成の様子を表したものである．収集した編集操作列及びそれらを階層的にグループ化する階層の節に対し，構成したいコミットの粒度を選択する為に階層の節を指定する．その後，指定された階層から順序関係を抽出し，その順序関係を利用して編集操作履歴の並び替えを行うことによって，ポリシーに従った粒度のコミットの構成を実現することができる．

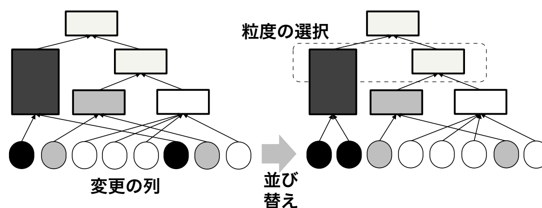

図 **4**　変更の再構成

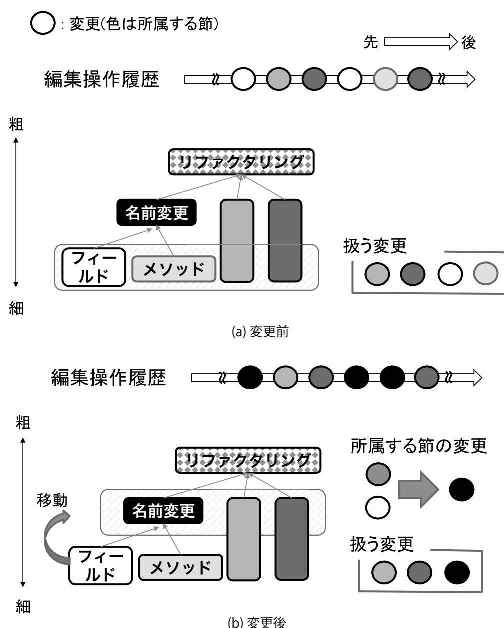

図 **5**　階層選択の変更の例

3.3.1　階層を構成する節の選択

ポリシーに従うコミットを構成する際，構成対象となる粒度を指定する必要がある．本手法において階層の選択は構成対象の粒度を選択することと同義である．階層の選択を変更することによって構成対象の粒度がどう変化するかを図 **5** を用いて説明する．図中の円は変更を表し，水平方向に伸びる矢印は右へ行くにつれて変更が新しいものであることを示す．いま編集操作履歴が存在し，その一部である変更 6 個からなる編集履歴列を切り出して考える．図 5(a) では 4 種類の階層を選択しており，それにより変更を 4 種類にグループ化している．そのうちフィールドに対する名前変更とメソッドに対する名前変更を異なる意味の変更として扱うことをやめ，より粒度の粗い，意味が抽象化された名前変更に基づくグループ化を考える．名前変更の節はフィールド及びメソッドに対する変更の 1 つ上の階層に属する．そのためフィールド及びメソッドに対する名前変更の階層の節に所属する変更を，図 5(b) のように，1 つ上の階層を構成している名前変更を表す節に所属が移動するよう変更する．この所属の変更により扱う変更が属する節は 3 種類に変化し，階層の再選択によって 2 つの異なる意味に分類されていた 3 個の変更が単一の意味の変更 3 個に変化したことが分かる．

© 2014 Information Processing Society of Japan

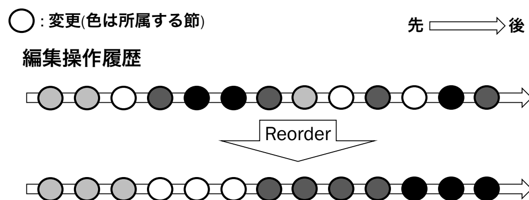

図 6　変更間に依存関係の存在しない編集操作履歴の並び替え

3.3.2　履歴の並び替え

階層的グループ化をされた編集操作履歴を整列させる履歴リファクタリングを考える．Historef には編集履歴を整列させる Reorder が定義されているが，既存のグループ間の順序関係を入力とし，既存グループの比較に用いる．本手法では末端階層間の順序関係を使用し，グループの比較には階層を用いる必要があるため，Reorder を末端階層間の順序関係を入力とし，比較に階層を用いるよう変更する．変更後の Reorder は以下になる．

名前　Reorder($H \equiv c_1 \cdots c_N, H' \equiv c_i \cdots c_j,$
$\preceq \subseteq T_{leaves} \times T_{leaves}$)

入力　編集履歴 H，H 中の部分列 H'，
末端階層間の順序関係 $\preceq$

操作　(1) $\mathrm{t(c_i) = t(c_k) \Rightarrow t(c_i) = t(c_j)}$ for all $\mathrm{i < j < k}$ 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($H, c_i \cdot c_j$) for all $i \cdots j$ s.t. $\mathrm{t(c_i)} = \cdots = \mathrm{t(c_j)}$.

ここでは Historef 上で定義された，与えられた 2 つの変更の順番を入れ替える Swap と与えられた変更列を単一の変更にまとめる Merge の 2 つの履歴リファクタリングを用いる．変更された Reorder を行うことにより，編集履歴 H' は整列され，分割 T' の各要素の集合に属している変更がそれぞれまとまりとなる．またそれぞれのまとまりの順序関係は $\preceq$ に従う．したがって与えられる分割 T' によって変更の粒度が細粒度から粗粒度まで変化することを表す．

図 6 は編集操作履歴 H に対して Reorder を行ったときの理想的な整列を表現したものである．垂直方向に伸びる青矢印は Reorder の実行を示し，その結果として同じ色の変更が近接するように整列されていることを表している．ただし変更間には新しい変更から古い変更に対して依存関係が成立することがあり，その場合は図 6 のようにはならない．変更間に依存関係が存在するとき，Reorder による整列は図 7 のように行われる．依存関係の方向は新しいものから古いものに限られ，依存関係の存在する 2 つの変更には Swap が適用できない．

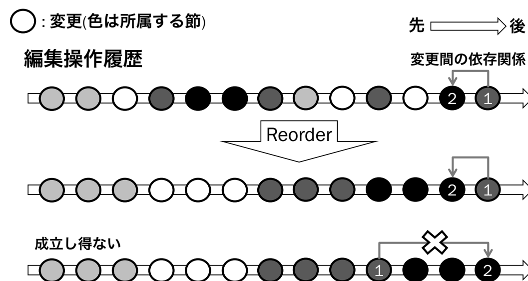

図 7　変更間に依存関係が存在する編集操作履歴の並び替え

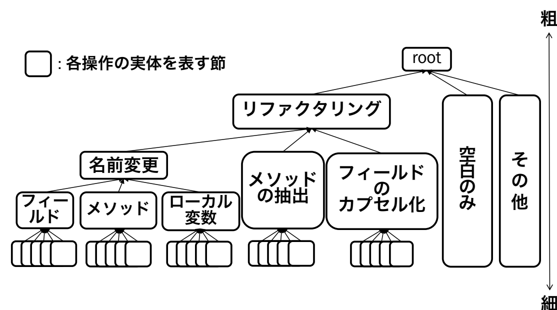

図 8　実装された階層の木構造

4.　実装

4.1　編集履歴の解析

提案手法で用いている OperationRecorder [15] を通して取得される編集操作履歴は，Eclipse IDE [1] が生成する編集操作オブジェクトから情報を取得することによって構成されている．OperationRecorder から Historef [13] へ編集操作が送信される際，編集操作に付加された情報に基づき，所属する末端階層を選択する．さらにその編集操作が追加する文字列，削除する文字列を解析することにより，節に付与すべきキャプション文字列を抽出する．

提案手法では，Eclipse によって提供されている編集操作のうち，使用頻度が高く，意味的な親子関係が存在すると考えられる 5 種類のリファクタリング操作をそれぞれ表現する節とした．それに加え，コードの整形に使用される空白の同時挿入を表現する節，他節に該当しない操作を表現する節からなる全 8 つの末端節を用意した．図 8 は生成される節を木構造で表現したものである．以下では図内の各階層の節の説明を行う．

根節はリファクタリング全体を表現するリファクタリング節，空白文字のみの編集を表現する節，その他を表現する節の 3 つの節とそれぞれ親子関係にある．空白文字のみの編集を表現する節及びその他を表現する節は末端階層に属する．

リファクタリング節は 3 種類のリファクタリング操作を表す節とそれぞれ親子関係にあり，それらは名前変更に関するリファクタリング全体を表す節，メソッドの抽出全体を表す節，フィールドのカプセル化全体を表す節である．

© 2014 Information Processing Society of Japan

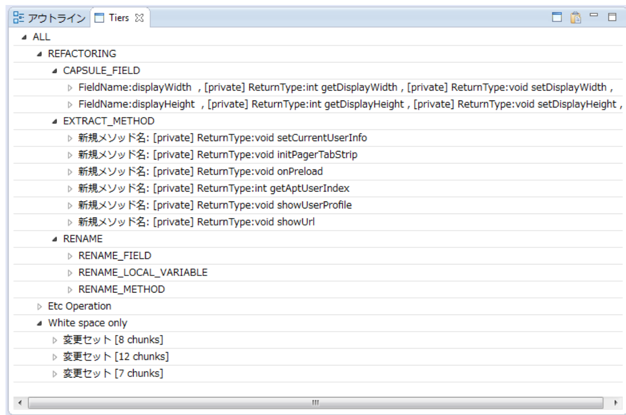

図 9　実装されたツールの画面例

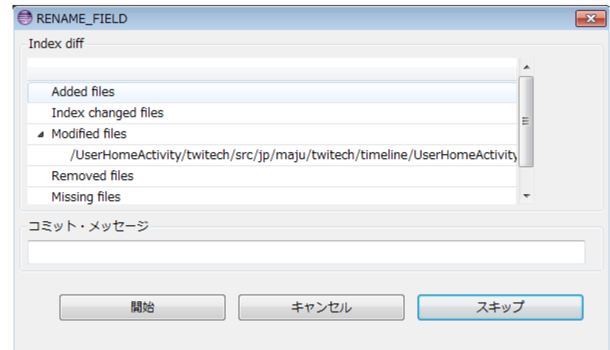

図 10　分割コミット用コミットダイアログ

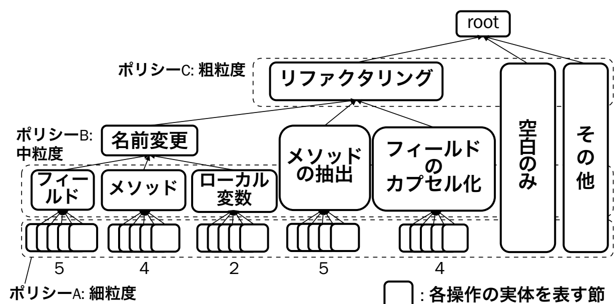

図 11　3 種類のポリシーと階層構造の様子

名前変更に関するリファクタリング節はさらに 3 種類の節とそれぞれ親子関係にある．それらは名前変更の対象によって区別され，フィールド名の変更全体を表現する節，メソッド名の変更全体を表現する節，ローカル変数名の変更全体を表現する節である．

また，5 種類のリファクタリング操作それぞれに対応する節は子となる節としてそれぞれの操作の実体を表現する節をもち，それら実体を表現する節は末端階層に属する．

4.2　階層構造の可視化機能

図 **9** に実装したツールの画面例を示す．これはリアルタイムでグループの階層構造を可視化したものであり， 2 回のフィールドのカプセル化，6 回のメソッドの抽出などを行った状態を表している．Eclipse の TreeView によって階層の木構造の形と種類が表示されており，各階層を構成する節は編集順序に従って上から下へ並んで表示されている．三角で表されたボタンをクリックして畳み込むことで子となる節や変更列を非表示にすることが可能であり， 図 9 では RENAME_FIELD などが畳み込みによって表示されていない． また末端階層に含まれる各種リファクタリング操作の実体は 4.1 節で記述した情報を開発者へ提示している．このビューは木構造の編集機能も持ち，分類の誤りの修正や新たな節の追加などの操作を必要に応じて行える．

4.3　分割コミット機能

ポリシー準拠のコミットを構成するにあたり，Git [2] との連係機能を実装し，分割コミットの支援も実現している．本手法による分割コミットを行う際に必要な手順は階層を構成する節の選択及び編集操作履歴の並び替えである．階層の構成要素である節を選択する際には，図 9 によって表示される節のうち， 指定したい節を表現した段を選択する必要がある． 選択された節が表現する粒度を構成対象の粒度であるとし，編集操作履歴の並び替えに必要な履歴内の分割点を算出する．算出された分割点を用いて分割ごとにコミットを行う．図 **10** はコミットごとに表示されるコミットメッセージ入力用のダイアログである． 上部に設置された Eclipse の TreeView に差分の生じたファイルが表示されており，コミットする変更がどの節の表現する意味かを表す文字列をタイトル部に提示する．図 10 は図 9 において RENAME_FIELD を表現する節を指定し，最初に表示されたものである．コミットダイアログによる操作を繰り返すことにより分割コミットを実現する．コミットダイアログにおいて，コミットのスキップは次のコミットのスカッシュに相応する操作を行う．キャンセルを選択すると分割コミット処理全体が中断され，履歴はロールバックされる．

5.　適用事例

5.1　適用対象

5.2　利用するコミットポリシー

Android 用 Twitter アプリケーションプロジェクトのある Java ファイルへの編集を対象として，提案手法を適用した．ファイルは 3903 行，フィールド 29 個，メソッド 105 個からなる．このファイルには重複したコードが多く見られ，また変数名がコーディング規約に従っていないなどの理由から，開発者は手動編集を含む大きなリファクタリングを行った．編集は約 40 分間にわたり，変更の数は 37 個，編集片にして 796 個存在した．このファイルに対して提案手法を実装した Historef を用いて編集操作履歴を記録し，3 種類のコミットポリシーそれぞれが期待する粒度での分割コミットを行った．また収集した編集操作履歴の内訳を下記に示す．

© 2014 Information Processing Society of Japan

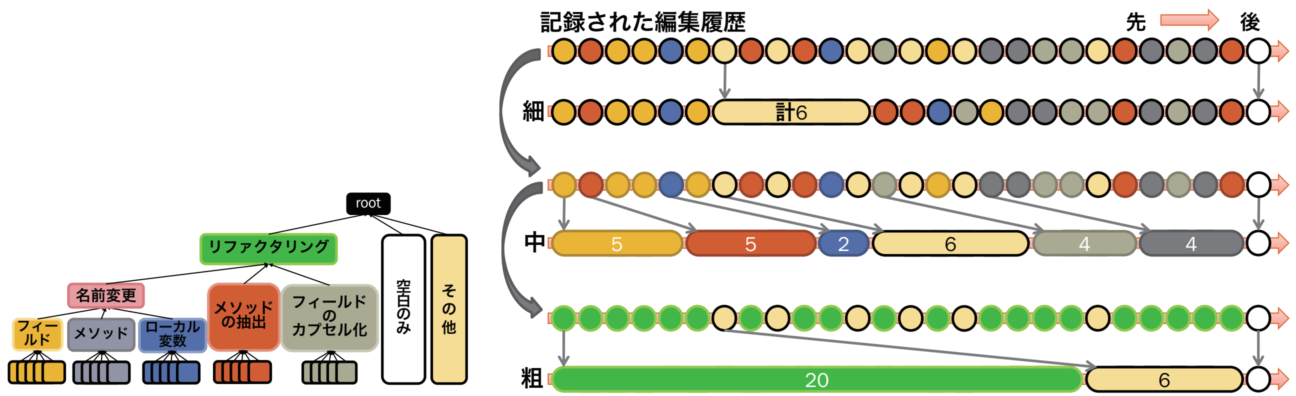

図 **12**　変更の再構成によるコミット構成の様子

- フィールド名の変更：5 回
- メソッド名の変更：4 回
- ローカル変数名の変更：2 回
- フィールドのカプセル化：4 回
- メソッドの抽出：5 回
- フォーマット操作による空白の編集：1 回
- その他の些細でない変更：6 回

図 **11** は本実験で利用するポリシーと階層の節の対応を表したものである．以下で 3 種類のポリシーをそれぞれ説明する．相対的に，ポリシー A が細粒度，ポリシー B が中程度の粒度，ポリシー C が粗粒度のものに相応する．

ポリシー A

差分理解を容易にするため，同種のリファクタリングであってもコミットを分け，またソースコードのフォーマットは差分の取得が難しいため分けるべきとするポリシーである．

定められる分割集合は，$T' = \{ \{$ フィールド名の変更 $1\}, \ldots, \{5\}, \{$ メソッド名の変更 $1\}, \ldots, \{4\}, \{$ ローカル変数名の変更 $1\}, \ldots, \{2\}, \{$ メソッドの抽出 $1\}, \ldots, \{5\}, \{$ フィールドのカプセル化 $1\}, \ldots, \{4\}, \{$ 空白文字のみ $\}, \{$ その他 $\} \}$ と表される．

ポリシー B

コミットの意味を，変更方法に応じて明確とするため，同種のリファクタリングはまとめてコミットを行うべきとするポリシーである．特に，名前変更に関するリファクタリングは対象に関わらずまとめて扱う．その他はポリシー A 同様，ソースコードのフォーマットを分離し，それ以外のプログラムの挙動を変化させない編集をひとまとめとする．

定められる分割集合は，$T' = \{ \{$ フィールド名の変更 $1, \ldots, 5$, メソッド名の変更 $1, \ldots, 4$, ローカル変数名の変更 $1, \ldots, 2\}, \{$ メソッドの抽出 $1\}, \ldots, \{5\}, \{$ フィールドのカプセル化 $1\}, \ldots, \{4\}, \{$ 空白文字のみ $\}, \{$ その他 $\} \}$ と表される．

表 **1**　ポリシーごとのコミット回数

ポリシー名	想定されるコミット回数	分割コミット回数
A	23	23
B	13	13
C	4	4

ポリシー C

リファクタリングなどのプログラムの挙動を変化させない編集とそうでないものを分けることで十分とするポリシーである．ただしソースコードのフォーマットに関しては，編集差分の観点から分ける．

定められる分割集合は，$T' = \{ \{$ フィールド名の変更の実体 $1, \ldots, 5$, メソッド名の変更 $1, \ldots, 4$, ローカル変数名の変更 $1, \ldots, 2$, メソッドの抽出 $1, \ldots, 5$, フィールドのカプセル化 $1, \ldots, 4\}, \{$ 空白文字のみ $\}, \{$ その他 $\} \}$ と表される．

5.3　適用結果

表 **1** は分割コミットによるポリシーごとのコミット回数と手法が期待するコミット回数の内訳である．図 **12** は適用実験によって編集操作履歴がどう並び替えられているかを示したものである．図中右にある変更の色は所属する節を表し，図中左の木構造の節の色と対応している．変更を表現する円の中に書いてある数字は変更内に所属する末端階層の節の合計数である．また灰色で表現された垂直方向に伸びる矢印は履歴の並び替えによる変更の移動のうち，順序関係を象徴する変更の移動を示している．垂直方向に並んだ編集操作履歴のうち，奇数番目の編集操作履歴は変更が所属している階層の節の様子を表している．偶数番目の編集操作履歴は実際に変更の再構成が行われた結果の様子を表している．

図 12 より，細粒度であるポリシー A は並び替えが少ないことが分かる．中粒度であるポリシー B では構成されるコミットが大きくなることが見て取れ，それに伴い並び替

© 2014 Information Processing Society of Japan

えが発生していることが分かる．粗粒度であるポリシーCは粒度が大きい分，並び替えが中粒度よりも少ないことが分かる．

5.4 結論

利用したコミットポリシー3種類について， それぞれ異なる粒度での分割コミットが可能であった． 実際に1コミットずつ変更内容の確認を行ったところ，全コミットにおいて，階層を構成している節が示す変更名と変更内容が一致していることが確認できた．また，それらのコミットがコミットポリシーにしたがった適切な粒度に再構成されていることも確認できた．これらのことは，前節で示したようなポリシーであれば提案手法を用いることにより自動的に分割が行えることを示している．また自動的な分割が可能であることから，コミットポリシーを定義されたフォーマットに従って記述することにより，プロジェクト内の開発者同士で共有する，あるいはポリシーの再利用が可能であることが分かる．

リファクタリングに関して， ポリシーAのコミットログは実際に行った操作履歴の流れとほぼ等しく，実際の変更と同様に5種類のリファクタリングが不規則に行われている． このことは，リファクタリング操作を行う度にコミットを行えば同様のコミット列を得られたことを示しているが，実際にこれを行うためには作業時間の関係から1分から2分に1回のペースでコミットを行う必要がある．しかし，これは開発者の負担や開発効率の低下の原因になると考える．

ポリシーBでは名前変更に関する変更がひとつにまとめられており， 名前変更に関して適切に整列されている．そのため，ポリシーAでは2回目と13回目のコミットとなっていたフィールド名の変更が近接している． これは，実際に開発段階でポリシーBに従おうとすると名前変更以外の5個の操作を後回しにする必要があることを示している． また，メソッドの抽出とフィールドのカプセル化に関してはそれらを近接させるような整列が行われていないが， 名前変更関連の変更が移動していることにより副次的に近くに配置されている．

3種類のポリシーに共通するコミットで重要なものに，空白文字のみに分類される変更がある． ソースコードのフォーマットを行うとき， 対象となった行はすべて変更として扱われるため， 版管理システムの差分表示を利用すると，空白文字のみの変更が発生した行と他の編集によって変更された行の区別を付けることが難しくなる． また，その影響範囲から競合の発生する可能性も高まる． そのためフォーマット操作が他の編集と混同してしまうと， 版間の差分の理解が困難になり， 競合が発生した場合はさらに開発者の負担が増えてしまう． このような開発者の負担になりやすい変更を分類し， 1つのコミットとすることが自動的に行えていることがわかる．

6. 関連研究

分割コミットに関する研究として，例えば以下のものがあるが，本稿とはアプローチの方針や目的が異なっている．

編集操作の分類や再構成． 本稿と同様に編集操作履歴を記録し，ファイルやメソッドなどの数種類の基準を用いて自動グループ化を行う研究がある [14]．この研究の目的は，手動でグループ化する必要がある部分を複数の基準を用いて自動化することで開発者の負担を低減させるものである．しかし，提案手法が目的とするコミットポリシーの粒度に対応できるグループ化を行うものではない点，編集操作の意味ではなく時間や編集箇所に着目している点で異なる．また，行ごとの編集操作を再構成することにより Task Level Commit の支援を行う手法もある [12] が，この手法も同様にリファクタリング操作などの意味的にまとまった変更を考慮していない点が異なる．

差分の分析による分割． 差分の分析を行うことにより，コミットを分割する既存手法が存在する [11]．しかしこの手法では，特にリファクタリングなどの版間からは抽出が困難な点に関して，制限が存在する．それに対し，本提案手法では開発者による一連の編集操作を履歴で保持することにより，開発者の意図や編集操作の意味を反映したコミットを構成できる．

差分を利用した意味混在の検出． 過去の編集差分を利用してテンプレートを作成し，現在の編集結果をテンプレートと比較することで意味の混在を検出する手法もある [8]．この手法では，現在の編集結果から典型的な意味の混在を検出することを目的としている．一方，我々の手法では，適切なコミットの粒度はプロジェクトが従うポリシーによって異なるとの考えに基づいており，複数の分割基準を扱える余地を与えている点が異なる．

7. おわりに

本稿では，一連の編集後でもポリシーに従ったコミットが構成できる手法を提案した．提案手法では開発者によるソースコードに対する編集操作を記録し，木構造を成すグループを用いて編集操作履歴をグループ化，その後編集操作履歴の列を並び替えることでコミットを再構成した．構成される木構造の節が構成されるコミットの粒度を表現するため，ポリシーから得られた適切な粒度を選択することでコミットの粒度を選択できる．そうして得られた区分から順序関係を算出し，整合性を取りながら編集操作履歴の並び替えを行った．節に利用するグループの基準として統合開発環境が提供する編集操作を用いて，提案手法の自動化ツールを Historef を拡張する形で実装した．また本稿では手法の有用性を確認するため，大きなリファクタリングを行った事例に対して，3種類のポリシーを用いて本手法

© 2014 Information Processing Society of Japan

を適用した．その結果，3 種類のポリシーがそれぞれ表現する粒度について，それぞれ適切にコミットを構成することができ，本手法の有用性を確認できた．

今後の課題を以下に示す．

新たな評価方法 本稿では正しくコミットが構成されたことのみを評価した．しかしながら実際に負担が軽減されたかどうかは被験者実験を行うなどにより更なる有用性を評価する必要があると考える．

階層を構成する節の追加 本稿で扱った対象以外にも，コミットポリシーの項目は一般的に存在する．代表的な項目である新機能の追加やバグ修正などは開発環境からは提供されない．変数間のデータ依存関係を用いてコミットを分割しようとする試み [11] 等の応用により，一部の自動化が可能であると考えている．

謝辞 本研究の一部は科研費（23700030）の助成を受けた．

参考文献

[1] Eclipse 4.3. http://www.eclipse.org/.
[2] Git - the fast version control system. http://www.git-scm.com/.
[3] Berczuk, S. and Appleton, B.: *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, Addison-Wesley (2002).
[4] Görg, C. and Weißgerber, P.: Detecting and Visualizing Refactorings from Software Archives, *Proc. 13th International Workshop on Program Comprehension*, pp. 205–214 (2005).
[5] Hayashi, S. and Saeki, M.: Recording Finer-grained Software Evolution with IDE: An Annotation-based Approach, *Proc. Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pp. 8–12 (2010).
[6] Herzig, K. and Zeller, A.: The Impact of Tangled Code Changes, *Proc. 10th International Workshop on Mining Software Repositories*, pp. 121–130 (2013).
[7] Kawrykow, D. and Robillard, M. P.: Non-essential Changes in Version Histories, *In Proc. 33rd International Conference on Software Engineering*, pp. 351–360 (2011).
[8] Kirinuki, H., Yoshiki, H., Keisuke, H. and Shinji, K.: Hey! Are You Committing Tangled Changes?, *Proc. the 22nd International Conference on Program Comprehension*, pp. 262–265 (2014).
[9] Murphy-Hill, E., Parnin, C. and Black, A. P.: How We Refactor, and How We Know It, *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 5–18 (2012).
[10] Negara, S., Vakilian, M., Chen, N., E.Johnson, R. and Dig, D.: Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?, *In Proc. 26th European Conference on Object-Oriented Programming*, pp. 79–103 (2012).
[11] 切貫弘之，堀田圭佑，肥後芳樹，楠本真二：ソースコード中の変数間のデータ依存関係を用いたコミットの分割，電子情報通信学会技術報告，Vol. 113, No. 269, pp. 67–72 (2013).
[12] 梅川晃一，井垣 宏，吉田則裕，井上克郎：細粒度作業履歴を用いた Task Level Commit 支援手法の提案，電子情報通信学会技術報告， Vol. 113, No. 489, pp. 61–66 (2013).
[13] 林 晋平，大森隆行，善明晃由，丸山勝久，佐伯元司：ソースコード編集履歴のリファクタリング手法，ソフトウェア工学の基礎 XVIII, 近代科学社，pp. 61–70 (2011).
[14] 星野大樹，林 晋平，佐伯元司：ソースコード編集操作の自動グループ化，コンピュータソフトウェア，Vol. 31, No. 3, pp. 277–283 (2014).
[15] 大森隆行，丸山勝久：開発者による情報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，Vol. 49, No. 7, pp. 2349–2359 (2008).

© 2014 Information Processing Society of Japan