社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# 細粒度作業履歴を用いたTask Level Commit支援手法の提案

梅川　晃一† 　井垣　　宏† 　吉田　則裕†† 　井上　克郎†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1 番 5 号
†† 奈良先端科学技術大学院大学
〒 630-0192　奈良県生駒市高山町 8916 番地の 5
E-mail: †{k-umekaw,igaki,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

**あらまし**　バージョン管理システム（VCS）を利用した開発では，単一のタスクに関連する小さな変更ごとにソースコードを VCS のリポジトリに格納（コミット）すること (Task Level Commit と呼ばれる) が望ましいとされている．一方で，コミット忘れや複数タスクの同時実施等により，1 回のコミットに複数のタスクに関連する大きな変更が含まれてしまうことがある．このような場合，どの変更がどのタスクに関連するかを開発者が把握しにくくなり，結果として変更内容の理解が難しくなることがある．そこで本稿では開発者がコミットを行う際に，ソースコードに対する変更内容をタスク単位で再構成できるよう支援する手法を提案する．提案手法によってタスクごとのコミットがどの程度実施できるようになるかを評価する実験を行ったところ，4 名の被験者全員について改善が見られた．
**キーワード**　バージョン管理システム，ソフトウェア構成管理，タスクレベルコミット

# Supporting Task Level Commit Pattern with using Fine-Grained Version History

Koichi UMEKAWA†, Hiroshi IGAKI†, Norihiro YOSHIDA††, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
Suita, Osaka 565-0871, Japan
†† Nara Institute of Science and Technology
Ikoma, Nara 630-0192, Japan
E-mail: †{k-umekaw,igaki,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

**Abstract**　In software development using Version Control System(VCS), developers are supposed to do a commit for every small change which is related to a single task (called Task Level Commit). However, a commit often includes the big change relevant to multiple tasks. In such a case, it becomes difficult to understand which changeset relates to which task. In this paper, we propose a Task Level Commit support system which enables developers to reconstruct their changesets relating to multiple tasks into small changesets relating to a single task.
**Key words**　Version Control System, Software Configuration Management, Task Level Commit

## 1. はじめに

複数人でのソフトウェア開発においては，ソースコードに対して誰がいつどのような変更を加えたかを記録しておくことが重要である．通常，このようなソースコードの変更履歴管理を目的として，バージョン管理システム (以下，VCS) [13] が用いられる．

VCS を利用した開発では，Task Level Commit(以下，TLC と呼ぶ) と呼ばれる構成管理パターンを順守することが重要であると言われている [7], [14]．TLC では，開発者は単一のタスクに関連する小さな変更ごとに，ソースコードを VCS のリポジトリに格納する（コミットと呼ばれる）べきであるとされている．TLC が順守されていると，VCS のリポジトリにコミットされたソースコードを見て，その変更がどのタスクに関連するものであるかを容易に理解できるようになる．

一方で実際のソフトウェア開発では，1 度のコミットに複数のタスクに関連する変更が含まれる場合がある [8]．1 度のコミットに複数のタスクに伴う変更内容が含まれると，どの変更

がどのタスクに関連するものであるかをリポジトリ内の変更履歴のみから判断することが難しくなることがある．そのような場合，特定のタスクに伴う変更内容を再利用したり，変更を差し戻したりといった対応が非常に困難になる．

そこで本研究では，Task Level Commit の順守支援を目的として，開発者によるソースコード編集作業後に，その編集履歴をタスク単位に再構成できるようにするシステムを提案する．本稿で提案する TLC 順守支援システムは以下の 3 つの手順により，編集内容の再構成を支援する．

A1: 開発者の細粒度作業履歴を自動的に取得する．
A2: 細粒度作業履歴を行単位の作業履歴に変換する．
A3: 履歴の統合，削除，並べ替えが可能な GUI を開発者に提示する．

本論文の以降の構成を以下に示す．2. 節では本手法の説明を行うにあたり，前提となる要素について説明を行う．3. 節では本手法における 3 つのアプローチを紹介し，それぞれについて具体的な処理を示す．4. 節では 3. 節の実装について紹介し，5. 節で評価実験について述べ，最後に 7. 節で本稿のまとめを行う．

## 2. 準　　備

### 2.1 バージョン管理システム

バージョン管理システム (Version Control System，以下 VCS) は，ソースコードの変更履歴を管理するシステムである [13]．

バージョン管理システムは，主として以下の役割を提供する．

- ファイルに対する作業を履歴として格納する
- 格納した履歴を開発者に提示する

ソフトウェア開発において，開発者は編集作業を行ったソースコードを VCS のリポジトリに格納（一般にコミットと呼ばれる）する．このとき，一度のコミットに含まれる，ソースコードに対して実施された変更の集合を changeset と呼ぶ．開発者によってコミットが行われると，コミットに含まれるソースコードにリビジョン番号と呼ばれる一意な数値や文字列が付与され，リポジトリに保存される．このリビジョン番号を利用することで，開発者はリポジトリに保存された過去のソースコードを取得（チェックアウトと呼ばれる）したり，特定のコミット間の差分を取得したり，特定のコミットで行われた変更を取り消したり，といったリポジトリに対する操作が可能となる．

### 2.2 Task Level Commit

Task Level Commit(以下，TLC) [7] とは “細かい粒度かつ一貫した単一のタスクごとにコミットを実行するべき” という考え方で，VCS の利用パターンの一つとして知られている．類似の VCS 利用パターンは多くの Open Source Software（OSS）開発においても採用されており，開発者に TLC を順守することを求めている [1] [2] [5]．

TLC が順守されていると，コミットに含まれる changeset と対応するタスクの関連が把握しやすくなる．すなわち，ソースコード中の特定の変更が，どのようなタスクの結果であるかを容易に把握できるようになる．また，VCS の特徴として，複数のコミットに含まれる changeset を統合することは比較的容易だが，1 つのコミットに含まれる changeset を複数に分割することは難しい．そのため，TLC の順守が求められる状況では，出来る限り細かい粒度でコミットが行われることが望ましいとされている．

### 2.3 tangled-changeset

実際の開発履歴には複数のタスクの実装を含む changeset が多く見られる [12]．また，ソースコードのリファクタリングや振る舞いに影響しない些細な変更が，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も報告されている [10] [11]．このような様々な変更が入り混じった changeset を tangled-changeset [9] と呼ぶ．tangled-changeset が発生すると，特定のタスクに対応する変更がどれであるかをリポジトリ内のコミット履歴から抽出することが難しくなる．結果として，変更内容が理解しにくくなり，あるタスクに対応する変更履歴の再利用/取り消しも困難になる．

## 3. 提 案 手 法

2.2 節において述べたように，粒度の細かい changeset を統合することは，大きい changeset を分割することと比較して容易である．そこで本稿では TLC の順守支援を目的として，開発者がコミットを行う際に，ソースコードに対する編集作業をタスク単位に再構成できるよう支援する手法を提案する．提案手法は細粒度開発履歴の自動保存と行単位履歴への変換，行単位履歴の再構成支援 GUI の提示というステップで TLC の順守支援を行う．順に詳細を述べる．

### 3.1 細粒度作業履歴の自動収集

我々が提案する手法では，実装中のソースコードを一定時間ごとに自動的に保存することで，より粒度の細かい作業履歴を取得する．このようにして保存された開発作業履歴のことを，本稿では細粒度作業履歴と呼ぶ．

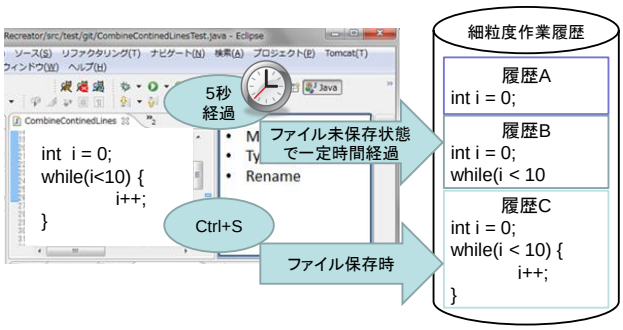


図 1　IDE による作業履歴の自動収集

細粒度作業履歴は図 1 に示すように，(1) 開発者が自分でファイルを保存したとき，(2) 編集作業後，開発者による保存が一定時間行われないとき，の 2 種類のタイミングで記録される．このようにして記録された細粒度作業履歴は図 1 のように行の途中で履歴が保存されることがありうる．我々の手法では，これ

らの履歴 1 つ 1 つを開発者が統合し，単一のタスクに対応する changeset として再構成を行う．しかしながら，細粒度作業履歴のような行の途中で保存されている履歴の場合，タスク単位での changeset の構築が困難になることがある．そのため，次のステップで，細粒度作業履歴を行単位の作業履歴に整形する．

### 3.2 細粒度作業履歴の再構築による行単位作業履歴の作成

行単位履歴の作成は以下の 2 ステップに分けられる．

- 細粒度作業履歴の分割
- 分割された細粒度作業履歴の統合

以降で，これらの各ステップについて説明を行う．

#### 3.2.1 細粒度作業履歴の分割

このステップでは，すべての細粒度作業履歴を，高々1 ファイルの 1 行のみの変更しか含まない履歴に分割する．

図 1 の履歴 B-C 間を分割する場合を例に，処理の概要を図 2 に示す．

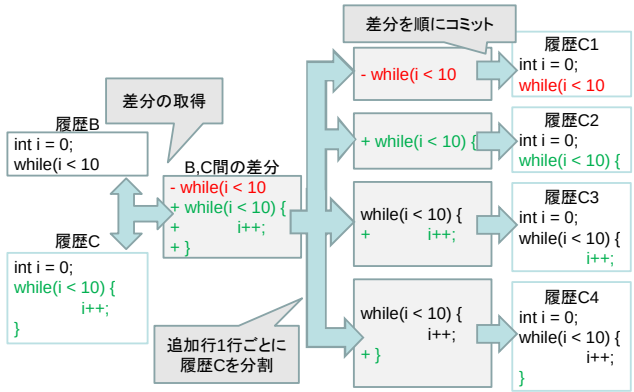


図 2　細粒度作業履歴の分割処理

まず，履歴 B と履歴 C の間の差分を取得する．差分は追加行 3 行と削除行 1 行で構成されている．

このような追加行と削除行で構成された履歴 B-C 間の差分を，1 行の追加/削除で構成された差分に分割する．分割された差分を履歴 B に対し順番に適用することで，履歴 B-C 間の差分が 1 行ずつ適用された履歴を取得することができる．結果として，細粒度作業履歴より粒度が細かい分割済み細粒度作業履歴が得られる．

上記の処理で作成された分割済み細粒度細粒度作業履歴は，複数行にまたがることが原則として無いため，履歴の粒度は常に行単位で一定である．しかし，行の途中での履歴が残ってしまうため，次のステップで履歴の統合を行う．

#### 3.2.2 分割済み細粒度作業履歴の統合

分割済み細粒度作業履歴が下記のパターンのいずれかに合致した時，複数の分割済み細粒度作業履歴を統合する．

（1） 変更内容が改行，空白，インデントのみのパターン

（2） 隣接した履歴が同ファイルを対象としており，同一内容を追加→削除しているパターン

パターン 1 の履歴は単独で意味のない履歴となる可能性が高いため，直後の履歴に統合する．パターン 2 は，記述した文字列を直後に削除した場合に加え，ある行の途中で履歴の保存が行われ，直後にその行に対し編集を加えた場合に発生する．発生する状況の例と処理の内容を図 3 に示す．

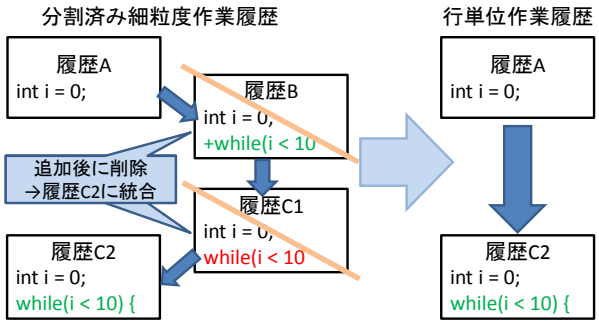


図 3　分割済み細粒度作業履歴の統合例

この例のように，同一行に対する履歴が連続して存在する場合，パターン 2 で示す統合処理が行われる．これらの分割・統合処理によって構成される行単位作業履歴には必ず 1 ファイルに対する 1 行の編集内容のみが含まれる (空行等を除く)．また，連続した履歴に，同一行に対する編集作業が含まれることは無い．

### 3.3 行単位作業履歴の統合，削除，並べ替え

これまでのステップで構築された行単位作業履歴を VCS の機能を用いて統合することで，タスク単位の changeset を構築することは可能である．一方で，複数のタスクが入り交じって開発されているようなケースでは，VCS の機能のみでは対応が困難な場合がある．そこで本研究では，図 4 に示すような行単位作業履歴の統合，削除，並べ替えに対応した GUI アプリケーションを構築する．

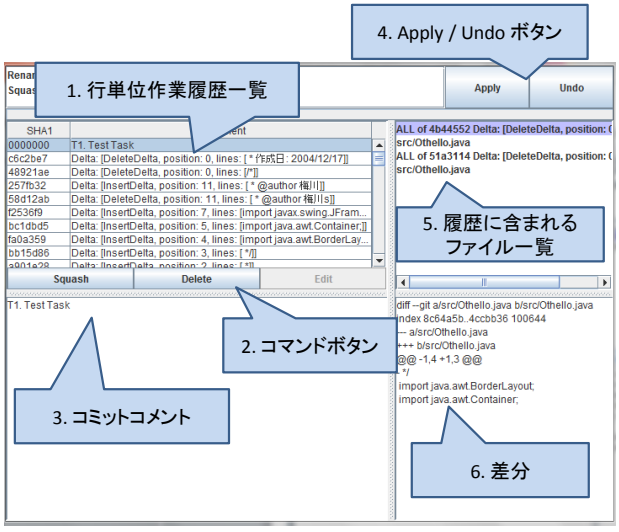


図 4　行単位作業履歴の統合，削除，並べ替えのための GUI アプリケーション

GUI の各構成要素について説明する．

- 行単位作業履歴一覧

行単位リポジトリ内の行単位作業履歴の一覧を表示する．一覧画面下部のコマンドボタン (後述) とあわせて，履歴の統合，削

除，並べ替えといった機能を提供する．一覧画面右および右下はそれぞれ選択した履歴に含まれるファイル一覧及び差分情報を提示する．

- コミットコメント

タスク単位の履歴を構築した後，コミット時コメントを付与することができる．

- Apply / Undo ボタン

コマンドボタン選択後，Apply を押すことで履歴の統合等の作業が処理される．Undo は実行した作業のやり直し．

ユーザは履歴を対象として下記 5 種類の操作を行うことができる．なお，下記操作は VCS の一つである Git の機能を一部利用して実現している．

（1） Squash

複数のコミットを統合する対象とする複数の連続した履歴を選択し，Squash ボタンをクリックすることで選択した履歴を一つにまとめることができる．

（2） Delete

対象とする履歴を選択し，Delete ボタンを押すことで対象の履歴を削除することができる．ここで “履歴を削除する” というのは，“その時点でのプロジェクトの状態の記録を削除する” ということである．そのため，削除した履歴より新しい履歴が存在していれば，削除したコミットで行われた変更は新しい履歴に含まれることになる．ただし，削除した履歴が最新のものである場合はそこで行われた変更そのものが削除される．

（3） Move

一覧画面上で履歴をドラッグ&ドロップすることで，順序を変更することができる．ただし VCS の機能では，同一ファイルへの編集内容を含む異なる履歴の順番を入れ替えることができない（Conflict が発生する）．そのため，自動的に Conflict を解消する仕組みを構築した．例えば，図 5 に示すような履歴 Z，A，B，C が存在するとする．このとき，VCS の機能のみで履歴 C を履歴 A の前に移動すると，履歴 A，B の整合性が保たれなくなり，Conflict が発生する．そこで我々は履歴の移動を差分の移動と捉え，履歴を履歴間の差分ごと任意の箇所に移動する仕組みを構築する．図 5 にある履歴 C を履歴 Z の後に移動する例にもとづいて，詳細な手順を述べる．ここで，c(X, Y) は Y から見た X との差分を表すものとする．履歴 C を履歴 Z の後に移動する場合，c(B,C) を履歴 Z に対して適用するものと考える．すなわち，変更前は履歴 Z に c(Z,A) が適用されていたところを，Z に c(B,C) をまず適用し，その後 c(Z,A) を適用，最後に c(A,B) を適用するという流れに変更する．この流れに沿って各履歴のソースファイルを変更することで，Conflict を発生させず，履歴の並べ替えを実施できるようになる．

（4） Rename

対象とする履歴を選択し，コミットコメント欄でコメントを編集することができる．

この GUI を用いることで，行単位作業履歴をタスク単位にまとめることが可能となる．

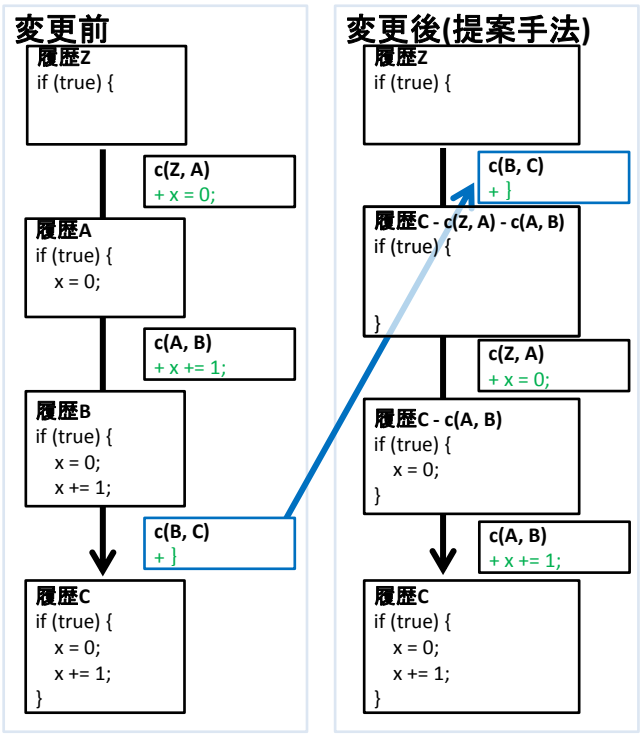


図 5 同一ファイルを対象とした履歴の並べ替えにおける Conflict 解消例

## 4. 実　　装

提案手法を実現したツールの実装について説明する．本ツールは，細粒度作業履歴収集ツール，行単位作業履歴構成ツール，行単位作業履歴操作ツールに分かれている．それぞれについて開発環境を以下に示す．

### 4.1 細粒度作業履歴収集ツール

細粒度作業履歴収集ツールは Java 言語で実装された Eclipse プラグイン “Save dirty Editor Eclipse Plugin (以下，SDE)” [6] に機能を追加する形で実装を行った．細粒度作業履歴収集ツールは開発者が成果物を格納しているワーキングディレクトリ，ローカルリポジトリの他にツール用のワーキングディレクトリ及びローカルリポジトリを管理しており，ユーザの開発にあわせて細粒度作業履歴をツール用のローカルリポジトリに格納する．

### 4.2 行単位作業履歴構成ツール

行単位履歴構成ツールは，細粒度リポジトリ内の細粒度作業履歴群を入力とし，行単位作業履歴群を含む行単位リポジトリを出力とする．行単位履歴構成ツールは Java プログラムとして実装されており，実装行数は約 2400 行である．

### 4.3 行単位作業履歴操作ツール

行単位作業履歴操作ツールは行単位リポジトリを入力として，ユーザに行単位作業履歴の操作が可能な GUI を提供する．本ツールは Scala により作成されたツールである “Uchronie” [3] に機能を追加する形で実装を行った．行単位作業履歴操作ツールは行単位リポジトリのコピーを生成し，そのコピーリポジ

トリ上で開発者が履歴の操作を行う．操作完了後に，開発者のローカルリポジトリに対してコミットを行う．

### 4.4 利用の流れ

細粒度作業履歴収集ツール，行単位作業履歴構成ツール，行単位作業履歴操作ツールの利用の流れを説明する．これらのツール群は，開発者が Git を用いて開発を行っていることを前提としている．細粒度作業履歴収集ツールは Eclipse プラグインとして実装されているため，利用時に開発者が意識することは無い．行単位作業履歴構成ツール及び行単位作業履歴操作ツールは，開発者自身がこの順に実行することで利用が可能である．

## 5. 評価実験

本論文で提案したツールを用いることで，TLC がどの程度順守できるようになるかを評価した．

### 5.1 実験概要

本実験では，被験者に詳細設計書にもとづいてソフトウェアの開発を行なってもらい，提案ツールを利用した場合と利用しなかった場合で TLC 順守率にどの程度差があるかを比較した．

本実験を行うにあたって，被験者には事前に TLC の考え方と，Git 公式ツールと提案ツールの利用方法のレクチャーを行った．被験者は大学院生 4 名で，全員が Java のプログラミング経験はあるが，Git の経験は無かった．Git 公式ツールのみを利用して提案ツールと同等の処理を実施してもらうことは困難であったため，基本的なコミット方法のみを指導し，実施してもらった．

実験では，被験者にソフトウェアの詳細設計書とタスクリストを提示した．被験者には仕様書に従ってソフトウェアを開発してもらい，成果物をタスクリストに記載されたタスク単位でコミットしてもらった．この作業を Git 公式ツールと提案ツールそれぞれで実施してもらい，TLC 順守率を比較した．

開発するソフトウェアは Java で作成されたオセロゲーム [4] に手を加えたものである．プログラムの規模は全体で 550 行ほどである．このオセロゲームの実装を 12 個のタスクに分け，前半と後半それぞれ 6 個ずつのタスクを 1 セットとして 2 つのタスクリストを作成した．

### 5.2 評価基準

各ツールを用いることでどれだけ TLC を順守することが出来たかを，全体のタスク数と TLC を順守できているタスク数の比率で比較し評価した．また，作業時間についても比較を行った．

この実験では，各タスクにおいて 1 つの changeset で機能が過不足なく実現されている場合に，TLC を順守できたと判断する．判断は著者らが目視で行った．その上で，タスク全体のうち TLC を順守できているタスクの比率を TLC 順守率として設定した．TLC 順守率は

$TLC$ 順守率 $= TLC$ が順守されたタスク数/全体のタスク数 $* 100(\%)$

という式で表される．

作業時間は細粒度作業履歴が記録され始めた時間から，被験者によって行単位履歴の操作が完了し，ローカルリポジトリにコミットが行われた時間までを計測している．Git 公式ツールの作業時間は実装開始時から最後のコミットが行われた時間までを計測している．

### 5.3 実験結果

TLC 順守率に関する結果を表 1 に示す．提案ツールを利用した場合の平均 TLC 順守率は 87.5%，公式ツールを利用した場合の TLC 順守率は 66.7%となった．

次に作業時間に関する結果を表 2 に示す．

その結果，提案ツールを利用した場合の作業時間が平均 76 分，公式ツールを利用した場合は平均 62.6 分となった．

## 6. 考　　察

Git 公式ツールを利用した場合に TLC が順守できなかった理由を被験者に確認したところ，以下の 2 つの理由が挙げられた．

- タスク終了時にコミットを忘れていた
- 機能を全て実装し終わる前にコミットしてしまった

これらの理由によるコミットミスは実際の開発でも同様に発生するものと思われる．また，同様に提案ツールを利用した場合に TLC が順守できなかった理由を被験者に確認したところ，行単位作業履歴とタスクの関連が分からなかったためであるとの回答が得られた．実際に被験者が提案ツール利用時にタスク割り当てを間違えた履歴例を図 6 に示す．この履歴では，public MainPanel というコンストラクタが追加されているが，被験者はこの変更をどのタスクに割り当てれば良いか分からなかったため，“タスク不明” としてコミットしていた．今回の実験では，提案ツールにおいて TLC が順守できなかった事例はすべてタスク割り当てが分からなかったことに起因していた．そのため，行単位作業履歴とタスクの関連付けが明確でありさえすれば，Git を利用したことがない開発者であっても，提案ツールを利用することで TLC を順守することは可能であるといえる．

作業時間については，異なるタスクリストを対象としていることと提案ツールと公式ツールで被験者が実施している作業内

表 1　TLC 順守率

| | 提案ツール (%) | Git のみ (%) |
|---|---|---|
| A | 100 | 66.7 |
| B | 100 | 83.3 |
| C | 66.7 | 50 |
| D | 83.3 | 66.7 |
| 平均 | 87.5 | 66.7 |

表 2　作業時間

| | 提案ツール (行単位履歴操作ツールのみ)(分) | 公式ツール (分) |
|---|---|---|
| A | 110(62) | 24 |
| B | 114(53) | 51 |
| C | 60(13) | 84 |
| D | 58(29) | 78 |
| 平均 | 76(39.3) | 62.6 |

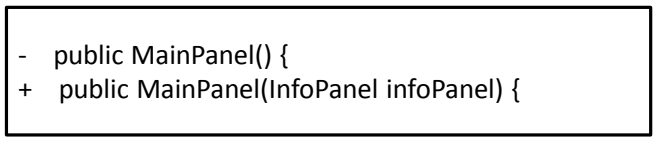


図 6　タスク割り当てを間違えた履歴例

容が異なるため，単純な比較に意味は無いが，行単位履歴操作ツールの利用に開発時間の半分程度の時間を要することが分かった．提案ツールの操作と利用時間について聞き取り調査を行ったところ，ツール自体は容易に利用することができたが，行単位作業履歴の粒度が細かすぎて，作業が煩雑になりがちであるとの回答が多かった．また，図 6 のような形式で表示される差分情報が分かりにくいとの感想も得られた．今後は行単位作業履歴の提示方法の改良や行単位だけでなく，ブロック単位等のもう少し大きい粒度での履歴提示等も検討していきたい．

また，3.3 節で述べた同一ファイルを対象とした履歴の並べ替えにおける Conflict 解消手法は，現時点ではまだあらゆる状況に対応できるとは言えない．例えば，履歴間の差分に削除を含む場合，正常に動作する場合としない場合とが考えられる．今後，より複雑なケースを対象とした Conflict 解消を実現していく予定である．

## 7. おわりに

本研究では，TLC の順守支援を目的として，IDE による作業履歴の自動収集，細粒度作業履歴の分割，統合による行単位作業履歴の作成，行単位作業履歴操作ツールによるタスクレベルでの履歴統合支援を実現した．

また実際に，細粒度作業履歴収集ツール，行単位作業履歴構成ツール，行単位作業履歴操作ツールとして実装を行い，粒度の細かい作業履歴の収集と提示，TLC の順守を支援できる環境を構築した．

評価実験では TLC 順守率が提案ツールの利用によって，平均 66.7%から 87.5%と改善がみられた．

今後の課題として，ブロック単位等の他の粒度での作業履歴の作成とタスク単位での履歴の再構成をよりやりやすくする GUI の改良を目指していきたい．

**謝辞**　本研究は，日本学術振興会科学研究費補助金若手研究 (B)(課題番号:24700030) の助成を得た．

## 文　献

[1] Commit policy - thymio & aseba. https://aseba.wikidot.com/asebacommitpolicy.

[2] Commit policy — qt wiki —qt project. http://qt-project.org/wiki/Commit_Policy.

[3] git rebase を決定的に刷新する最強ツール uchronie をリリースしました. http://tomykaira.hatenablog.com/entry/2013/07/13/222203(非公開).

[4] Java でゲーム作りますが何か? - 人工知能に関する断創録. http://aidiary.hatenablog.com/entry/20040918/1251373370.

[5] Policies/commit policy - kde techbase. http://techbase.kde.org/Policies/Commit_Policy.

[6] Save dirty editor eclipse plugin. http://savedirtyeditor.sourceforge.net/.

[7] Stephen P Berczuk and Brad Appleton. *Software configuration management patterns: effective teamwork, practical integration*. Addison-Wesley Professional, 2003.

[8] Shinpei Hayashi and Motoshi Saeki. Recording finer-grained software evolution with ide: An annotation-based approach. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pp. 8–12, New York, NY, USA, 2010. ACM.

[9] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pp. 121–130. IEEE Press, 2013.

[10] David Kawrykow and Martin P Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 351–360. ACM, 2011.

[11] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, Vol. 38, No. 1, pp. 5–18, 2012.

[12] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP 2012–Object-Oriented Programming*, pp. 79–103. Springer, 2012.

[13] 岩松信洋, 上川純一ほか. Git によるバージョン管理. 株式会社オーム社, 2011.

[14] 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-第 18 回ソフトウェア工学の基礎ワークショップ (FOSE 2011) 予稿集, pp. 61–70, 2011.