修士学位論文

題目

細粒度作業履歴の分割，統合による Task Level Commit 支援手法の提案

指導教員
井上 克郎 教授

報告者
梅川 晃一

平成 26 年 2 月 5 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 25 年度 修士学位論文

細粒度作業履歴の分割，統合による Task Level Commit 支援手法の提案

梅川 晃一

内容梗概

ソフトウェア開発においてソースコードへの変更履歴を管理することを目的として，バージョン管理システム (VCS) が用いられている．VCS を用いることで，開発者は過去の変更内容の取り出しや変更差分の参照が可能となる．

実際のソフトウェア開発において，VCS に記録された一度のコミットに複数の変更が含まれる場合がある．一度のコミットに複数の変更が含まれると，変更とタスクの関係性の理解や，特定のタスクに伴うファイル変更内容を再利用したり，変更を削除したりといった対応が非常に困難になる．そこで複数の変更が含まれたコミットを発生するのを防ぐため，Task Level Commit が提案されている．Task Level Commit は，単一のタスクに関連する小さな変更ごとにコミットするべきという考え方である．Task Level Commit を順守することでコミット内の変更の理解や利用が容易になる．しかし，様々な理由で Task Level Commit の順守が失敗する可能性がある．

そこで本論文では Task Level Commit に準拠した履歴の再構築を支援するための手法を提案した．キーアイディアとして作業履歴の自動収集と行単位作業履歴の作成，行単位作業履歴の作成と表示，Task Level Commit のための行単位作業履歴の統合についての説明を行った．また，本手法を実現するためにツールの開発も行った．

その上で，開発したツールによって Task Level Commit をどの程度支援できるかを評価するための実験を行った．その結果，Task Level Commit 順守率が向上し，Task Level Commit の順守支援に有効であることを確認した．

主な用語

バージョン管理システム
Task Level Commit
ソフトウェア構成管理

目 次

1 はじめに

ソフトウェア規模の巨大化に比例して，開発に関わる人員数もソースコードの量も増大している．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非常に重要である．通常，このようなソースコードへの変更履歴を管理することを目的として，バージョン管理システム (以下，VCS)[32] が用いられている．VCS は開発者によるファイルへの変更を記録し，変更履歴として管理する．VCS を用いることで，開発者は過去の変更内容の取り出しや変更差分の参照が可能となる．一般に，開発者がファイルへの変更を VCS に記録することをコミットすると言う．コミット行動では，ファイルの変更内容や開発者情報，日時，変更内容についてのコメントといった情報が VCS に登録される．このような開発者のコミット行動によって保存される情報のことをコミットと呼ぶ．コミットの情報を確認することで，開発者はソフトウェアの変更の推移を理解することが出来る．さらに，ファイル変更内容を他のプロジェクトで再利用したり，特定の変更内容を削除したりといった VCS の活用が実際に行われている．

ソフトウェア開発プロジェクトでは，機能の追加やバグ修正といったタスクが開発者に与えられる．開発者は与えられたタスクに従って開発を進める．このとき，開発者が各タスクを完了するたびに，そのタスクの遂行に伴うファイル変更内容が VCS にコミットされることが望ましいとされている．

一方で，実際のソフトウェア開発においては，一度のコミットに複数の変更が含まれる場合がある [20]．

これは以下のような場合に発生する．

- 一つの変更を終了した後にコミットを行わず次の変更を行なってしまった場合
- 変数のリネームやタイプミス修正などの細かい変更が組み合わされた場合
- 複数の変更が関係性を持つため個別のコミットが難しい場合

1 度のコミットに複数のタスクに伴うファイル変更内容が含まれると，どの変更がどのタスクによるものであるかを開発者が判断することが難しくなる可能性がある．結果として，特定のタスクに伴うファイル変更内容を再利用したり，変更を削除したりといった対応は非常に困難になる．

複数の変更が含まれたコミットを発生するのを防ぐため，Task Level Commit[15] が提案されている．Task Level Commit は，単一のタスクに関連する小さな変更ごとにコミットするべきという考え方である．Task Level Commit を順守することでコミットが持つ変更の目的と変更されたコードの対応付けが簡単になる．そのため，タスクと変更の関連を理解しやすくなり，タスクに伴う変更の再利用や削除が容易となる．

しかし，実際の開発では様々な原因により複数種類の変更が同時に行われ，Task Level Commit の順守に失敗する可能性がある．そのため，開発の流れに関わらず TLC の順守を行うことができる環境が必要であると考えた．

そこで本研究では，Task Level Commit の順守を支援する手法を提案する．具体的には，以下の 3 つのキーアイディアで手法を実現する．

K1 IDE による細粒度作業履歴の自動収集

K2 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

K3 手作業による行単位作業履歴のタスクレベルでの統合支援

また，本手法を実装した 3 つのツールの実装についても説明を行う．

加えて，被験者 4 人を対象に本論文で提案したツールと Git の公式ツールの比較実験を行った．実験では，被験者にコーディングを行ってもらい，成果物を Task Level Commit を順守しつつコミットを行ってもらった．その上で，Task Level Commit を順守できていたか，作業時間，使用感の 3 つの面で比較を行った．

本論文の以降の構成を以下に示す．2 章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 章では本手法における 3 つのキーアイディアを紹介し，各キーアイディについて具体的なアルゴリズムを示す．4 章で 3 章のアルゴリズムの実装方法を紹介する．5 章で本手法の効果を測るための評価実験について，6 章で関連する研究について紹介する．最後に 7 章で本論文のまとめを行う．

2　準備

2.1　ソフトウェア構成管理

ソフトウェア構成管理 (Software Configuration Management．以下，構成管理) とはソフトウェアの開発や保守の管理技術である [31]．構成管理は変更が発生する生産形態における管理技術であり，混乱した生産活動を支援する手法と管理の枠組みである．

ソフトウェアの品質向上を目指して，開発手順を明確化したものを “開発プロセス” と呼び．構成管理は “開発プロセス” の一要素として定義されている．学術的な定義から言えば，構成管理は

- 履歴管理
- 品質管理

の 2 つの要素で成り立っている [33]．

この 2 つのうち，品質管理には，実際に品質を検証するための “試験” という要素が不可欠である．また，構成管理と “試験” は “開発プロセス” の上では個別に定義されているため，一般的に単独で構成管理といった場合，履歴管理のことを指す．
履歴管理では表 1 のような情報を記録しておき，必要な時に必要な情報を適宜取り出して利用する．

履歴管理を行うことで，以下のメリットが得られる．

- 成果物のバックアップをとることができる
- 間違った変更をしてしまった場合に，簡単に以前の状態に戻すことができる
- 問題が発生した場合，変更経緯をたどることで，顕在化した時点を特定できる

表 1: VCS が保存する情報

(物理的な)When	何時の作業か？
(論理的な)When/Where	どの時点の成果を元にした作業か？
Who	誰の作業か？
What	どのファイルに対する作業か？
Why	作業理由は？ ※ 正しくメッセージを記述した場合
How	どのような作業か？

- 2 つの時点の成果を比較することで，その間での成果量を算出できる

2.2 章以降，履歴管理を行うための機構であるバージョン管理システムの概要，その中で用いられる changeset および tangled-changeset の概念について説明する．

2.2 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，ファイルの履歴を管理するシステムである．バージョン管理システムは，主として以下の 3 つの役割を提供する．

- ファイルに対して施された追加・削除・変更などの作業を履歴として蓄積する
- 蓄積した履歴を開発者に提供する
- 蓄積した履歴を編集する

VCS は開発者に対し．何らかの作業によって生成されたファイルについての複数の履歴 (バージョン) を記録する．そして，後から古い履歴の取り出しや，差分の参照ができる仕組みを提供する．このファイルの履歴が保存された場所をリポジトリ，履歴の一つ一つをコミットと呼び，履歴をリポジトリに追加することをコミットするという．逆にリポジトリからコミットされたファイルを取り出し，ローカルに保存することをチェックアウトと呼ぶ．また VCS によっては，ファイルの削除や移動の履歴を確認する機能や，特定の利用者がファイルの占有権を獲得するロック機能，複数の変更を統合するマージ機能がある．VCS は，複数人で同一のファイルを編集する必要がある時に有効である．例えば，VCS を使わずテキストファイルをファイルサーバーにおいて共有していたとする．この状態で編集やコピーを続けていくと，誰が最新版を持っているかや，誰がいつどこを編集したかなどの履歴が把握することが難しくなる．このような場合に，このファイルを VCS で管理すれば，変更点と変更者，変更日時を管理できるようになる．

VCS はリポジトリの構成とファイルの管理方法から 3 つに分けることができる．個別 VCS，クライアント・サーバー型の集中型 VCS，そして Git[4] に代表される分散型 VCS である．VCS の種類と，代表する VCS ソフトウェアを表 2 に示す．

ここでは Subversion に代表される集中型 VCS と，本研究で用いた Git が含まれる分散型 VCS について，詳細の説明を行う [32]．

- 集中型 VCS(図 1)
  集中型 VCS はクライアント・サーバー型の構成をとり，リポジトリはサーバー上に格納されている．リポジトリに対する操作はクライアントから行うが，ネットワーク的に接続していなければならないという制約がある．集中型 VCS としては CVS のほか

に Subversion や商業ソフトウェアである Perforce などがある．これらは現在でも多く利用されているが，次第に分散型 VCS に移行しつつある．

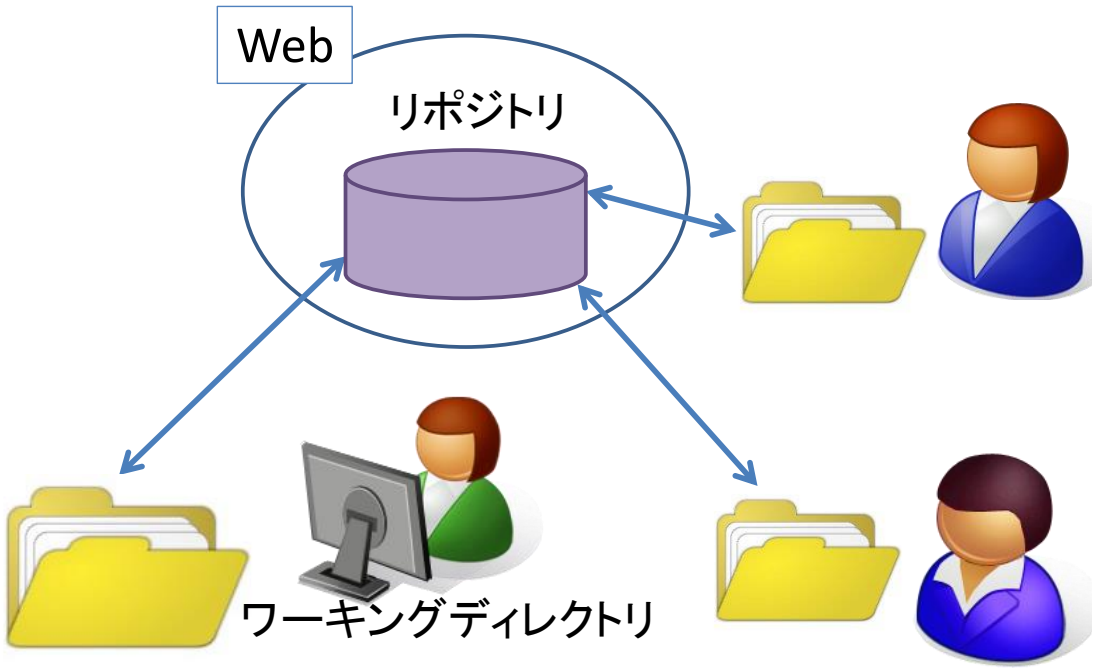

図 1: 集中型 VCS の概要図

- 分散型 VCS(図 2)
  分散型 VCS では，リポジトリをサーバー上だけではなく，クライアントであるローカル上にも作成する．コミットの参照や差分を取得する場合に，手元にあるリポジトリにアクセスするため，場所や時間，ネットワーク接続如何にかかわらず VCS を利用することができるという利点がある．また，ローカルファイルシステムにリポジトリ

表 2: VCS の種類と概要

VCS の種類	概要 (ツール例)
個別 VCS	ファイル単位で個別システムでバージョン管理を行う (SCCS，RCS)
集中型 VCS	リポジトリをサーバーで一元管理し，コミットなどの操作はクライアントから行う (CSV，Subversion，Perforcz)
分散型 VCS	リポジトリをクライアントでも管理する．このクライアント間でリポジトリの連携ができる．(Git，Bazaar，Mercurial，BitKeeper)

があるので高速に動作することも利点の一つである．近年では，オープンソースプロジェクトの多くで Git をはじめ Bazzar や Mercurial などの分散型 VCS が採用されている．

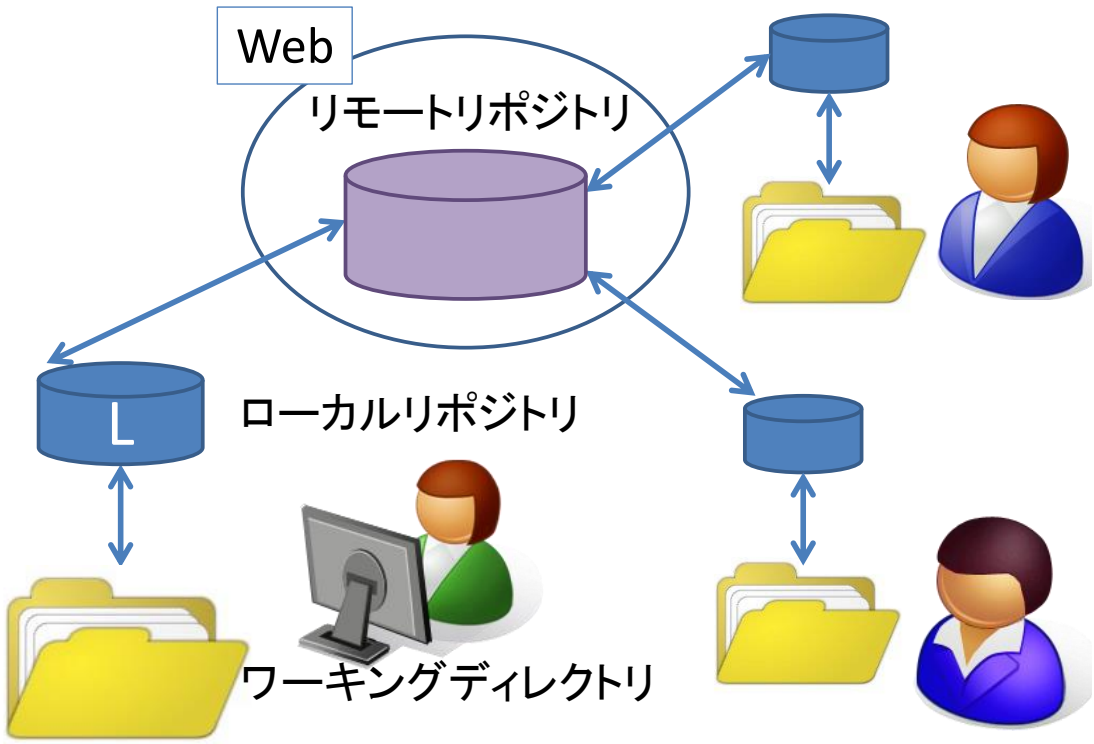

図 2: 分散型 VCS の概要図

次に，多くの VCS で変更の管理に用いられる changeset の概念について説明する．

2.2.1 changeset

近年のバージョン管理システムでは，複数のファイルに対する変更を記録する際に，“誰が，どの時点の状態に対して，いつ，どのように” 変更したのかを，個別のファイルごとではなく，全てのファイルを横断的に一括して記録している．この記録された内容を “コミットに含まれた変更全体の集合” という意味で changeset と呼ぶ．changeset は記録の際の不可分な単位であるため，初期状態から順に changeset を積み上げることで，ある変更を実施した時点での全てのファイルの状態を完全に再現することができる．また，過去に行ったソフトウェアへの変更を再利用，あるいは取り消す際には changeset 単位で操作を行うことになる．そのため，複数の変更が changeset に存在していると，コミットの内容を理解すること

や，変更の再利用などが難しくなる可能性がある．そのため，VCS の利用者は一度のコミットに含めるタスクを出来るだけ少数，理想的には一つのみにするべきであるとされている．

しかし，実際の現場における開発履歴には複数のタスクに由来する changeset が多く見られる [20]．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も報告されている [22] [25]．このような，様々な変更が入り混じった changeset を tangled-changeset と呼ぶ．

2.2.2 tangled-changeset

tangled-changeset(以下，t-changeset)[21] とは，機能追加や，バグ修正，リファクタリングなど，複数のタスクに関連する変更が混在する changeset のことを指す．

開発者はタスクが完了すると，ソースコードへの変更を他の開発者と共有するため VCS へコミットする．このときに開発者が複数の変更をまとめてコミットすることで，t-changeset が発生する．

t-changeset が発生すると，単一のタスクに対応する変更を VCS から抽出することが難しくなるため，結果としてあるタスクに対応する変更の再利用/取り消しが難しくなる．そのため，t-changeset の発生は極力避けるべきだと考えられている．

VCS において，t-changeset をコミットしてはいけない理由として，changeset は再利用のしやすさ，理解しやすさ，変更の削除のしやすさに影響を与えるということがある．

- 再利用のしやすさ
  もし，t-changeset から変更の一部をピックアップしたい場合，changeset から望む変更を抽出する必要がある．
- 理解しやすさ
  コミットログに複数のタスクが記述してある場合，それぞれのタスクの記述とソースコードの対応を組み合わせることが出来ず，changeset に対する理解が難しくなる．
- 変更の削除のしやすさ
  changeset の一部を削除しようとすると，changeset 全体を削除してしまう．

この章で，t-changeset の問題点を例とともに示し，また t-changeset が問題となる原因について説明する．

例として，開発者がソースコードに対してバグ修正などの特定の目的をもって変更を行う状況を考える．開発者はリポジトリからチェックアウトしたソースコードに対しバグ修正を行い，それをリポジトリにコミットする．この時，開発者がタイプミスなどの自分のタスク

に直接関わりのない修正すべき点を見つけた場合，自分のタスクであるバグ修正だけでなく，また，変数名の変更などの簡単なタスクも同時に行われる可能性がある．

仮に開発者が本来のタスクであるバグ修正に加え，タイプミスの修正や変数名の変更などの簡単なタスクも同時に行い，それらの変更を一度にコミットすると，バグ修正とタイプミス修正，そして変数名の変更という 3 つのタスクが含まれた t-changeset が発生する．

このような独立したタスク同士であれば個別にコミットすることも簡単である．しかし，あるタスクと別のタスクが関連するものである場合，それらを一つ一つ個別にコミットすることは難しくなる．複数のタスクが関連を持つ例を図 3 に示す．

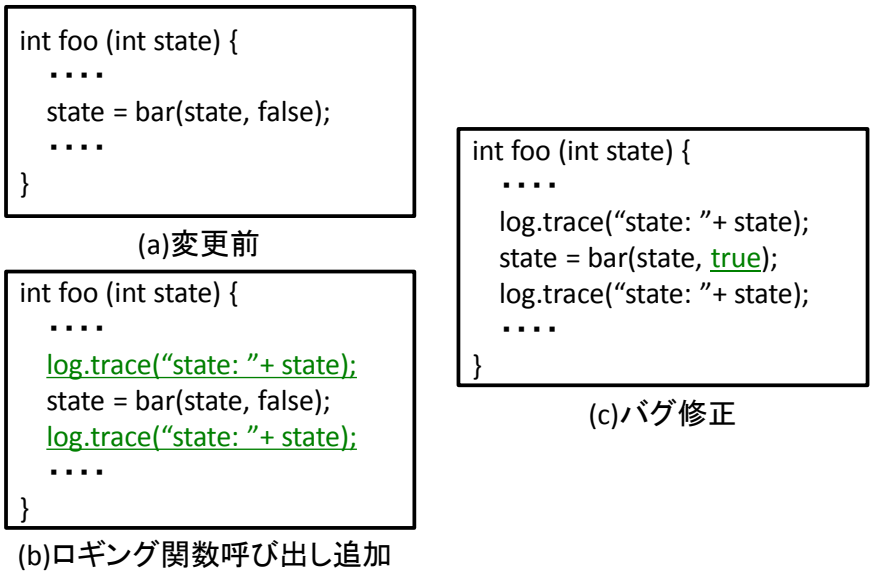

図 3: 複数のタスクが関連を持つ例

ここでは，ある開発者が図 3(a) のコード中にあるバグを修正する場面を考える．このソースコード中に存在するバグは，関数 bar の第二引数が本来 true であるべきところを false に指定している，というものである．開発者はロギング関数をソースコードの複数箇所に挿入することで，バグの発生箇所を特定した (図 3(b))．そのあとで開発者はバグを修正する変更をコードに加えた (図 3(c))．これらの変更によって，プログラムは正常な動作を行うようになった．

このとき，図 3(c) にあるように，コードにはロギング関数の追加 ((a) → (b)) とバグ修正 ((b) → (c)) という 2 つの変更が含まれる．コミットポリシーによってはロギング関数の追加をバグ修正とは別のコミットにするべき，あるいは今後の開発に使わない関数として削除するべき，と定められている場合がある．別コミットにする場合，実際は図 3(b) の時点で

コミットすることは難しい．なぜならロギング関数の追加が十分に行われたと判断できるのは，バグ修正が終了したと確認が出来た段階であり，ロギング関数の追加とバグ修正は同時に行われるからである．よってこのふたつの変更がそれぞれ別のタスクとして扱われる場合においても，通常の VCS の利用法では 2 つの変更を個別にコミットするのは難しい．

また，ロギング関数を今後の開発に使わない関数として削除する場合，2 つのタスクが含まれる t-changeset からロギング関数をバグ修正と分けて削除することは難しい．これは，VCS が changeset を不可分の単位として扱う関係上，ロギング関数を削除する場合に同じ t-changeset に含まれるバグ修正もともに削除されてしまうためである．

同様に，large refactorings[18] [24] も t-changeset を発生させる．large refactorings とは複数の基本的なリファクタリングを組み合わせたものである．large refactorings の中で一つのリファクタリングが失敗した場合，そのリファクタリングが行われる前に戻す必要がある．しかし，t-changeset の中に含まれる large refactorings から基本的なリファクタリングを分けることは難しい．

Git や Darcs[3] など，最近の VCS は changeset の一部を選択して新たなコミットを構築しなおす機能を備えている．しかし，これらの VCS は行が隣接するコード群を一つの変更として扱うため，コード群に複数のタスクが含まれている場合，この機能は十分に動作しない．

Operation-based アプリケーション [27][29] は，より細粒度なバージョン管理機能を提供する．しかし，これらのアプローチは開発者の意図を問わず，完全に自動化されている．そのため，開発者が意図する通りの適切な changeset を構築できない可能性がある．Commit 2.0[16] では，2 バージョン間の changeset を集約することが出来る．しかし，changeset を untangled な変更に分割することはできない．OperationRecorder[20] は，開発者がツールに対してタスクの区切りをコールすることで，開発者の意図どおりかつ自動的に changeset を分割する機能を提供している．しかし，上記で説明した状況において，必ずしもツールに対してタスクの区切りをコールできない，あるいはコールを忘れる可能性があるという問題がある．

2.3 構成管理パターン

構成管理を行うにあたって，開発チームの参加者全員が t-changeset をコミットしてはいけない，などの開発プロセスにおける一貫した考え方を共有している必要がある．ソフトウェア開発環境において実行される全ての開発プロセスの中で，ソースコード，実行可能形式，バージョン管理システムなどは状況にかかわらず同じ動作を行う．そして，それを扱う開発者の振る舞いによって開発プロセスの品質が変化する．開発者の振る舞いによって品質が変化する開発プロセスに対し，ツールや環境によって開発者の振る舞いを支援する．また，手作業によるプロセスには動機付けを行うことで開発者に一貫した行動を取るように促す．

このツールによる開発者の振る舞いの支援，作業に対する動機づけの手段，および支援や動機付けの手段を行使するシチュエーションをパターンとしてまとめたものが構成管理パターンである．構成管理パターン自体には自明かつ簡単なものが多い．しかし，そのパターンをどのように適用するかの詳細は必ずしも自明ではなく，その適用の詳細がパターンの価値である．プロセスにパターンを適用する際に重要な点について Alexander はこう述べている [13].

“私達にそれまで知らなかったことを見せてくれるという理由では，大したことはありません．しかしそのかわり，(パターンの適用は) 既に知っていること，特に幼稚に見えたり原始的に見えたりするがために私達が認めていなかったことを明らかにするという点で重要なのです”.

次に，構成管理パターンの一つである Task Level Commit について説明する．

2.3.1 Task Level Commit

Task Level Commit(以下，TLC) とは “細かい粒度かつ一貫したタスクごとにコミットを実行するべき” という履歴管理におけるパターンである．タスクの種類としては,

- 新しい機能の追加
- 障害レポートの解決
- リファクタリング

が考えられる．書籍 [15] で示されている適度な変更量を有するタスクの例を示す.

- 障害レポート (大きな問題は 2 つ以上のコミットにまたがる可能性がある).
- 廃止予定の関数呼び出しから，システム全体で使う新しい API 呼び出しに変更すること.
- 廃止予定の関数呼び出しを，それと密接に関連する，システムの一部のために変更すること.
- 一日のうちに達成した一連の変更.

TLC においては，複数のコミットを統合することは出来るが，1 つのコミットを複数に分割することは難しい．また changeset のサイズが小さければ周囲への影響が小さくなることから，なるべく細かくコミットするべきと考えるる．また，一つのコミットに複数のタスクを含めて良い例外パターンとして 1,2 行のコードが多くの障害レポートに紐付けられている場合などを定めている.

2.3.2 OSS で用いられるコミットポリシー

それぞれのプロジェクトでコミットに関する独自のポリシーが設定されており，コミットとタスクの関連に関しても記述されている．実際の OSS で用いられているコミットポリシーのうち，コミットの粒度に関する記述の例を示す.

- KDE TechBase[11]
  git has the ability to commit more than one file at a time. Therefore, please commit all related changes in multiple files, even if they span over multiple directories at the same time in the same commit. This way, you ensure that git stays in a compilable state before and after the commit, that the commit history (svn log) is more helpful and that the kde-commits mailing list is not flooded with useless mails.

  OTOH, commits should be preferably “atomic” - not splittable. That means that every bugfix, feature, refactoring or reformatting should go into an own commit. This, too, improves the readability of the history. Additionally, it makes porting changes between branches (cherry-picking) and finding faulty commits (by bisecting) simpler. While most KDE developers do not pay much attention to that rule, some do so very much. Consequently, you should be very careful about it when making commits in areas which you do not maintain.

- Qt[2]
  Make atomic commits. That means that each commit should contain exactly one self-contained change - do not mix unrelated changes, and do not create inconsistent states. Never “ hide ” unrelated fixes in bigger commits. Make coding style fixes only in exactly the lines which contain the functional changes, and comment fixes only when they relate to the change - the rest is for a separate commit.

- Thymio & Aseba[1]
  Your commits should be organized in a logical set of atomically small changes. Commit as often as possible.

これらのコミットポリシーで言及されているのは,

- 機能追加，バグ修正，リファクタリングなどは個別にコミットすること
- 関係のない変更を混同せず，纏まった単位でコミットすること
- 極力小さな単位でコミットすること

- コミットの頻度を極力上げること

である．直接的に TLC についての言及は存在しないが，方向性は TLC に限りなく近く設定されている．

また，総じてコミットの具体的な粒度については言及が見られず，OSS や人員によってその粒度は変化すると考えられる．そのため様々な環境にまたがってソフトウェア開発におけるタスクの粒度を一意に決定することは難しい．

2.3.3 TLC の問題点

TLC を実現することで，開発履歴の可読性や利便性が増す．そして，それぞれの開発現場ではコミットポリシーを設定し，可読性や利便性の高い開発履歴を蓄積しようとしている．しかし，実際のソフトウェア開発の現場では，コミットポリシーを設定しているにも関わらず開発履歴の中に多くの t-changeset が存在する．その原因として考えられるのが，

- 作業が終了した時にコミットを忘れたまま次の作業をはじめた
- あるタスクを実行しているときに，そのタスクを完了するために必要な別のタスクが発生した

などである．上記の原因はソフトウェア開発においてたびたび発生するため，一般的な VCS において TLC を実現し続けるためには開発履歴自体の修正が必要となる．また，タスクとタスクの間にコミットを挟むことが出来るパターンであっても，TLC に作業のプロセスを縛られ，タイプミス修正などを後回しにしてしまうと，作業の見落としや効率の悪化に繋がると考えられる．

2.4 本研究の目的

ここで，2 章の総括と VCS の問題，研究の目的について説明する．

ソフトウェア開発において VCS を用いる場合，開発履歴の可読性や利便性を高めるため一つのコミットには一つのタスクに由来する変更のみを含め，TLC を順守するべきである．実際に，開発現場ではコミットポリシーを設定し，TLC を順守を順守しようとしている．しかし，ソフトウェア開発における開発履歴の中には多くの t-changeset が存在する．原因として，コミット忘れや実行中のタスクを完了するのに他のタスクが必要になる，などの回避できない状況があり，また VCS の機能だけでは t-changeset を分割することが難しいからであると考えられる．

これらの問題を解決するため，本研究では開発中に自動で収集した編集内容から粒度の細かい作業履歴を提供し，それらを統合することで TLC の順守を支援する手法を提案する．

3　提案手法

3 章では VCS の抱える問題の解決策として，開発履歴の自動保存と行単位への分割と統合，断片化した開発履歴の統合を行うことで TLC を支援する手法を提案する．

3.1　キーアイディア

本論文における提案手法を実現するために, 以下の 3 つのキーアイディアを採用する．

K1 IDE による細粒度作業履歴の自動収集

K2 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

K3 手作業による行単位作業履歴のタスクレベルでの統合支援

この手法は，開発者がソースコードに与えた変更を細粒度作業履歴として全て記録する．その上で，記録した細粒度作業履歴を可読性が高く，なおかつ複数のタスクが含まれる可能性の低い行単位に分割し，行単位作業履歴として記録し直す．最終的に記録された行単位作業履歴を開発者に提示し，開発者が望む粒度のタスクに統合する支援を行う．

以降，この手法の全体像と各キーアイディアを実現するためのアルゴリズムについて説明を行う

3.2　全体像

図 4 にキーアイディアの実現を行ったシステムの全体像を示す．

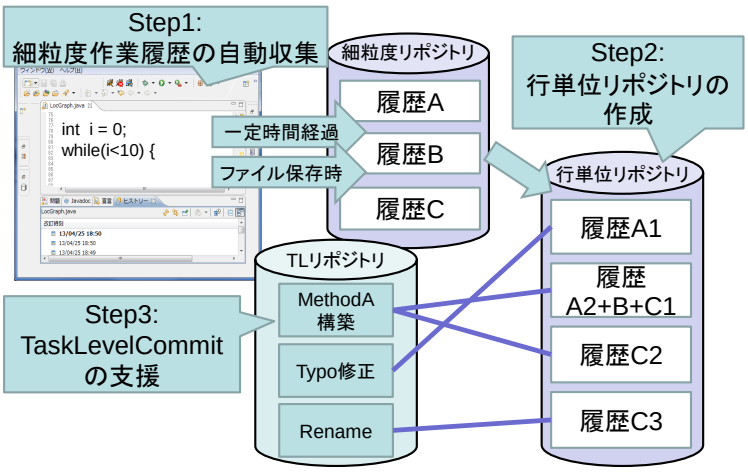

図 4: 提案手法の全体像

以降で,K1 を実現するための IDE を用いた開発ログの自動収集手法と, K2 の細粒度作業履歴を行単位履歴へ分割，統合する手法，K3 を開発者に提供するための GUI および内部処理について説明を行う．

3.3 K1 : IDE による細粒度作業履歴の自動収集

バージョン管理システムは一般的に開発者がソースコードに対して行った変更が全て記録されると考えられている．しかし，実際には開発に関するいくつかの情報が失われている．

2.2.2 章で説明したロギング関数などの，後の作業で必要がなくコミット前に削除されたコードや，図 5 にあるような，変更の上書きなどが実際の開発において VCS では記録が出来ない情報である．

Negera らの研究 [26] によると，VCS に保存されている changeset からはコードへの変更全体の 37%が失われているとされている．また，コードの変更順序も VCS には記録されない情報である．

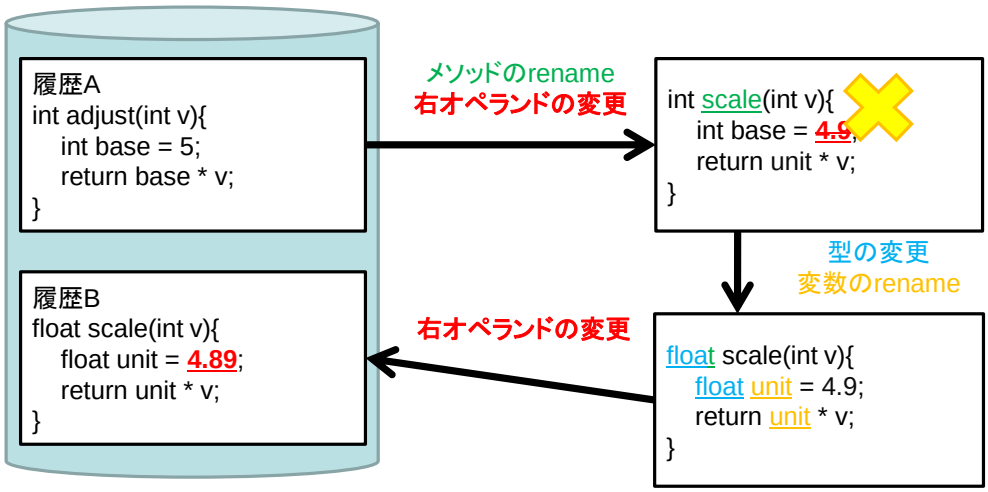

図 5: 失われる情報の例

changeset を複数のコミットとして再構成してコミットするにあたり，失われた 37%の変更がタスクにとって重要である可能性がある．また，一つのタスクは一度に纏めて行われると考えられるため，コードの変更順序も記録されることが望ましい．

そこで，K1 では IDE を用いて一定時間ごとにプロジェクトの状態を自動的に保存する．K1 によって記録された作業履歴を，本論文では細粒度作業履歴と呼称する．また，細粒度作業履歴を保存するリポジトリを細粒度リポジトリと呼称する．具体的には,IDE がプロジェ

クト内にあるファイルの内容が変更され, かつその変更が保存されていない状態にあるとき, プロジェクトの状態を一定時間ごとに保存する．また，ファイルが保存されたタイミングに同じくプロジェクトの状態を保存する．これらの方法で細粒度作業履歴を自動的かつ定期的に収集することが可能となる．ツールの機能を図 6 に示す.

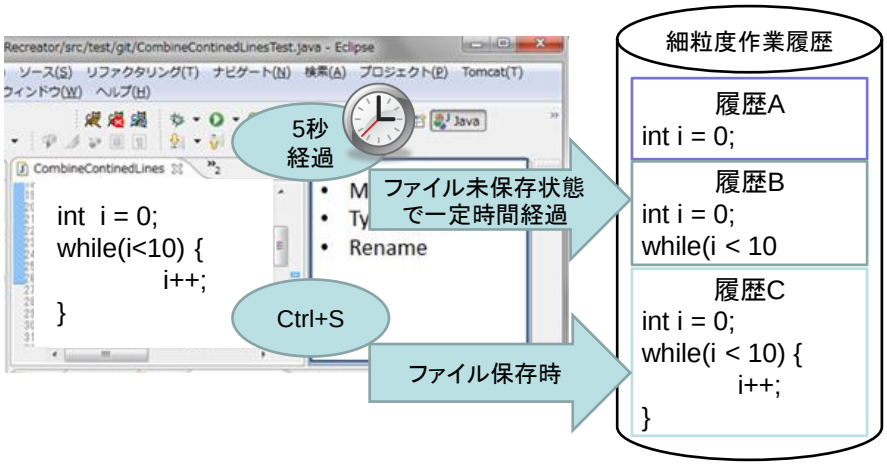

図 6: K1:IDE による作業履歴の自動収集

K1 においては履歴が自動で保存されるため，図 6 の履歴のように関数を分断した形で changeset が記録される場合がある．このままでは可読性も利便性も低くなってしまうため, ソースコード中の関数などを分断しない開発履歴として再構成する必要がある．K2 ではこの細粒度作業履歴をソースコードを分断しない開発履歴として再構成する.

3.4 K2 : 細粒度作業履歴の分割，統合による行単位リポジトリの作成

K1 によって短いスパンで開発の推移を記録した細粒度作業履歴を記録することができた. しかし，K1 では細粒度作業履歴を自動的に収集するため，一部がソースコード中の関数などを分断する形で保存されている．そのため，細粒度作業履歴をさらに粒度を高めた上で, 関数や文などを分断しない形に整形する．ソースコード中の関数や文を分断しないソースコードの単位として，粒度の大きなものから順に以下の単位が考えられる.

- ブロック単位
- ステートメント単位

- 行単位
- 関数単位

手法の目的から，構成要素に複数のタスクが含まれる可能性が低いことが最優先とし，また可読性が高いことから行単位での再構成を選択した．本論文では，行単位に再構成された開発履歴を行単位作業履歴，行単位作業履歴を記録したリポジトリを行単位リポジトリと呼ぶ．

行単位履歴の作成は以下の 2 ステップに分けられる．

K2-1 細粒度作業履歴の分割

K2-2 分割済み細粒度作業履歴の統合

以降で，これらの各ステップのアルゴリズムについて説明を行う．

3.4.1 K2-1 : 細粒度作業履歴の分割

このステップでは細粒度作業履歴の粒度をより細かくするため，細粒度作業履歴を行単位で分割．この処理によって，最大でも 1 ファイルに対する 1 行のみの changeset をもつ分割済み細粒度作業履歴が得られる．

処理の概要を，図 6 の履歴 B-C 間の changeset を分割する場合を例にとり，図 7 に示す．

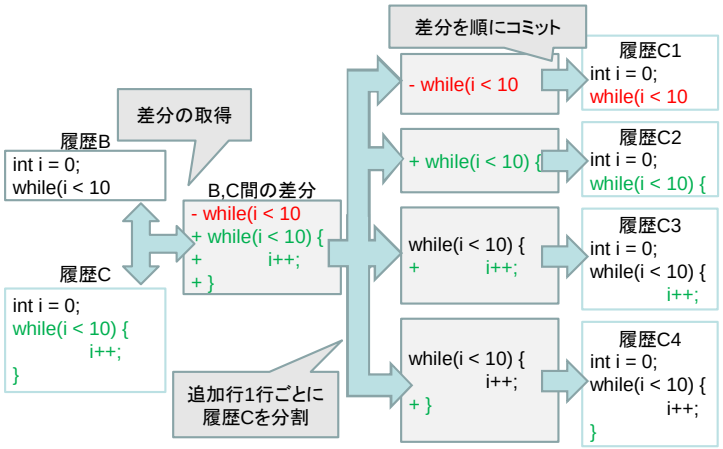

図 7: K2-1:細粒度作業履歴の分割処理

まず，履歴 B と履歴 C の間の差分，つまり履歴 C の changeset を取得する．差分は図 8 と図 9 のような，追加行と削除行で構成されている．

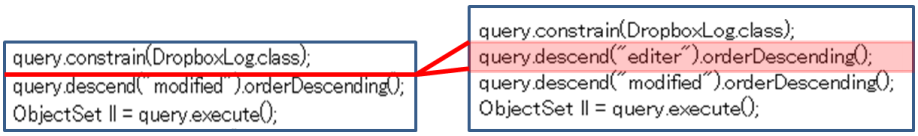

図 8: “行の追加” の例

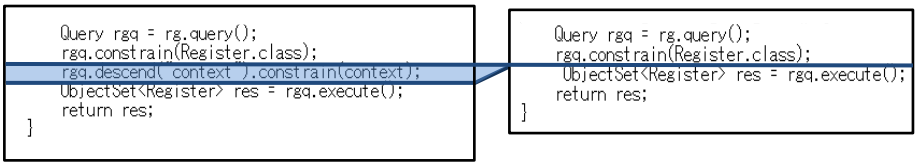

図 9: “行の削除” の例

分割によって得られた追加行と削除行で構成された履歴 B-C 間の差分を，一行の追加/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 行ずつ適用されたファイルの内容を取得することができる．取得したファイルの内容を順番にコミットすることで，細粒度作業履歴より粒度がより細かい分割済み細粒度作業履歴が得られる．

この処理の例外として，一つの細粒度作業履歴で一定以上の行数のコードが扱われる場合，K2 の処理は行わない．大量のコードの自動生成やテンプレートのコピー&ペースト，あるいはその削除など，コーディング中に大量のコードを纏めて扱うことがある．その際に扱ったコードを行単位に分割すると大量の作業履歴が生成されてしまう．そのため，本処理においては行数の閾値を設定し，細粒度作業履歴内の changeset の行数が閾値以上であれば，分割処理は行わないよう設定した．

分割済み細粒度細粒度作業履歴は，複数行にまたがる changeset が存在せず行単位で粒度が一定である．しかし，依然としてソースコードの構成要素を分断する履歴が存在している．また，空行の追加などのソースコードとして意味を持たない changeset のみを持つ履歴が存在する．余計な履歴の数が増えると可読性，利便性の面でマイナスとなるため次のステップで履歴の統合を行い，これらをリポジトリから排除する．

3.4.2　K2-2 : 分割済み細粒度作業履歴の統合

K2-1 で作成された分割済み細粒度作業履歴は上記の理由から，可読性，利便性が十分ではない．そこで，分割済み細粒度作業履歴がいくつかのパターンに合致した時，複数の分割済み細粒度作業履歴を統合する．統合が行われるパターンは以下の 2 つである．

1. changeset 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への changeset を持ち，同じ内容を追加→削除しているパターン

1. のパターンの changeset は実装の上で意味のない履歴となってしまうため，可読性を上げるためリポジトリから省く必要がある．そのため，直後の履歴に統合する．2. のパターンは，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に示す．

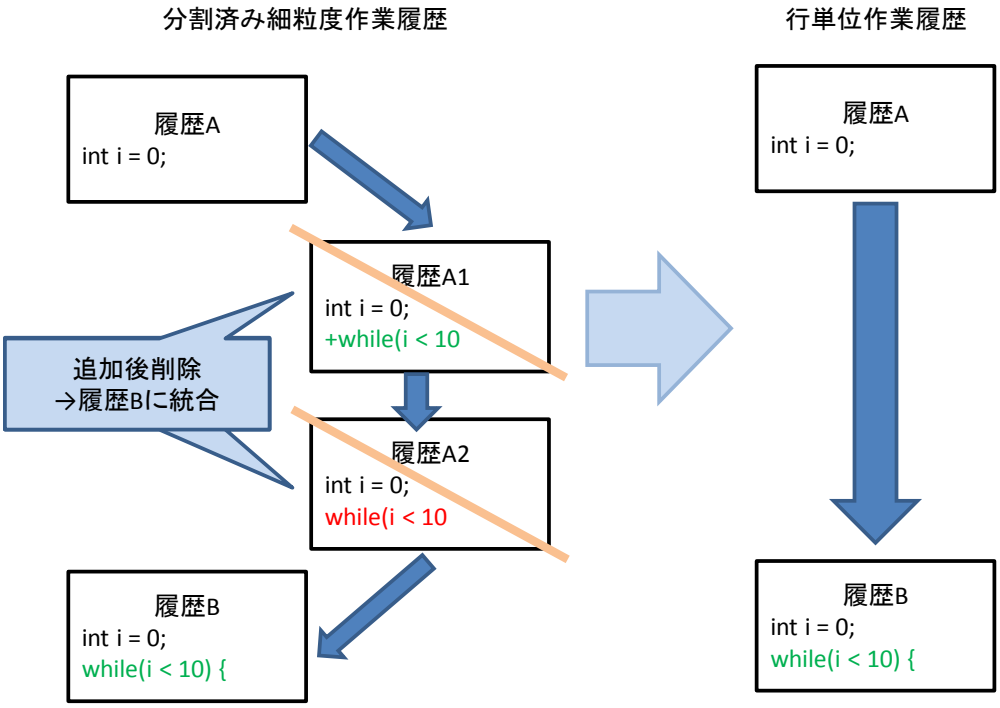

図 10: 分割済み細粒度作業履歴の統合例

このようなパターンを統合することによって，行の途中で分断された履歴を省き，可読性，利便性を向上することができる．

以上の処理で細粒度作業履歴を行単位作業履歴に再構成する．行単位作業履歴には必ず 1 ファイルに対する 1 行の changeset のみが含まれる．

3.5 K3 : Task Level Commit の支援

K2 の処理が終了した時点で，開発者は行単位リポジトリから行単位作業履歴を確認することができる．TLC を実現するためには，これらの行単位作業履歴をタスク単位に統合する必要がある．複数の作業履歴を統合する機能は VCS によって提供されている．しかし，それらの機能を十分に扱うには多くの知識を必要とする．また，コマンドベースでの作業となるため，統合作業中の状況を開発者が把握することが難しい．その上，統合作業に失敗した場合に作業の巻き戻しをする際も別の知識が必要となってしまう．

そこで，K3 では VCS の機能を GUI で扱うことができるアプリケーションを提案し，行単位作業履歴の閲覧，統合を支援し TLC 実現の支援とする．

図 11 に提案する GUI と機能の概要を示す．

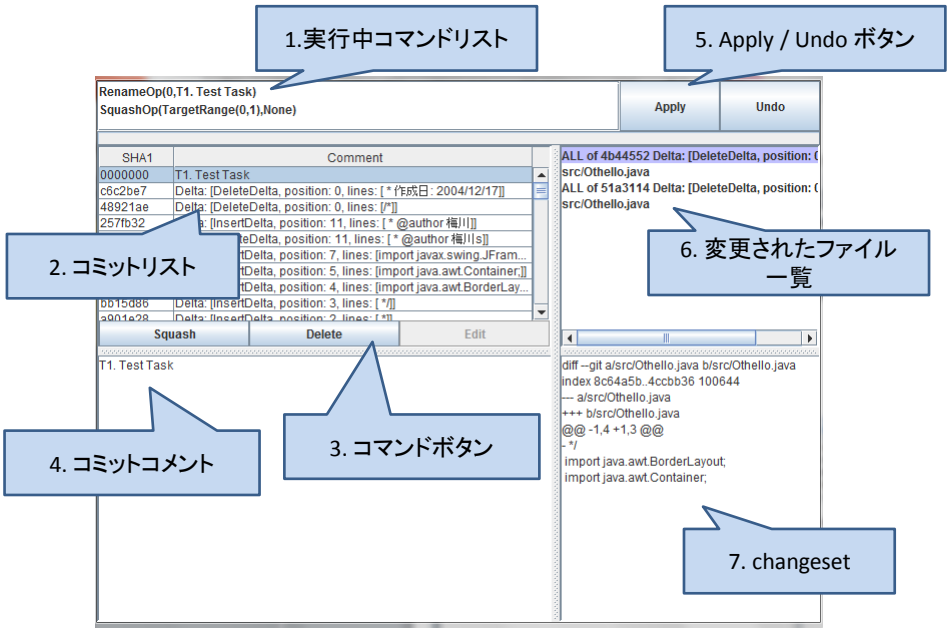

図 11: K3 : GUI

提案する GUI の機能一覧を示す．

1. 実行準備コマンドリスト
   GUI によって実行されようとしているコマンドのリスト

2. コミットリスト
   行単位リポジトリ内の行単位作業履歴の一覧を表示

3. コマンドボタン
   それぞれのボタンに対応するコマンドを実行準備

4. コミットコメント
   コミットリストで選択したコミットのコメントを表示

5. Apply / Undo ボタン
   実行準備状態のコマンドを Apply ボタンで全て実行，Undo ボタンで 1 つずつ取り消し

6. 変更されたファイル一覧
   コミットリストで選択したコミットで変更されたファイルの一覧を表示

7. changeset
   コミットリストで選択されたコミットにおいて，変更されたファイル一覧で選択されたファイルに関する changeset を表示

コマンドは以下の 4 種類である．

- DeleteOp
  対象とするコミットを選択し，Delete ボタンを押すことで対象のコミットを削除する．ここで “コミットを削除する” というのは，“対象のコミットで行われた変更を削除する” のではなく，“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，削除したコミットより新しいコミットが存在していれば，削除したコミットで行われた変更は新しいコミットの中に存在している．ただし，削除したコミットが最新のものである場合はそのコミットで行った変更を他のコミットが持っていないため，変更自体も削除される．

- MoveOp
  コミットを移動する．コミットリスト上でコミットをドラッグ&ドロップすることで，コミットの順序を変更することが出来る．

- RenameOp
  対象とするコミットを選択し，コミットコメント欄でコメントを編集する．

- SquashOp
  複数のコミットを統合する対象とする複数のコミットを選択し，Squash ボタンをクリックすることで選択したコミットを一つのコミットとして集約する．

このGUIを用いて開発者は行単位作業履歴を行ったタスクごとに集約する．そして，集約したコミットのコミットコメントを設定することで，TLCを実現したコミットがリポジトリの中に作成される．

以上のK1からK3までの手法を用いることによって，開発者はコーディングの最中にTLCやコミットポリシーに縛られることなく，望んだ粒度でコミットを行うことが出来るようになる．

4　ツールの実装

4 章では，3 章で提案した手法を実現したツールの実装について説明する．ツールは，K1 を実現した細粒度作業履歴収集プラグイン，K2 を実現した行単位作業履歴ツール，K3 を実現した TLC 支援ツールに分かれている．そこで，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プラグインの実装

細粒度作業履歴収集プラグインは開発者のファイルに対する変更を入力とし，細粒度作業履歴を出力とする．細粒度作業履歴収集プラグインは Java ソースコードにより作成された Eclipse プラグイン “Save dirty Editor Eclipse Plugin (以下，SDE)”[12] に機能を追加する形で実装を行った．以下．細粒度細粒度作業履歴収集プラグインを “プラグイン” と呼ぶ．

4.1.1　開発環境

開発言語は Java，Eclipse Kepler 64bit 上で開発を行った．また，VCS として分散型 VCS である Git を用いた．Java から Git の各コマンドを実行するためのライブラリとして，JGit[10]，および Gitblit[6] を利用した．以降で説明するツールも，同様に VCS として Git を用いている．

4.1.2　処理の流れ

SDE は，コーディング中に以下の 2 つの動作を行う．

- 保存されていないファイルの状態を一定時間ごとに “ファイル名. 拡張子.snapshot” 形式でバックアップする
- ファイルを保存した時に対応した.snapshot ファイルを削除する

このバックアップとバックアップファイルを削除するタイミングで，編集中のファイルをプラグイン用ディレクトリ内にコピー，プラグイン用ディレクトリの状態を Git にコミットすることで作業履歴を記録したリポジトリを作成する．ファイル未保存時，保存時のプラグインの処理を図 12，13 に示す．

プラグイン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，専用のディレクトリ (SDE 用ディレクトリ)，リポジトリ (SDE 用リポジトリ) を持つ．

まず，ファイル未保存時の処理を説明する．プラグインで設定した一定時間ごとに IDE 上で現在アクティブかつ未保存状態のファイルの中身を.snapshot 形式のファイルとしてバッ

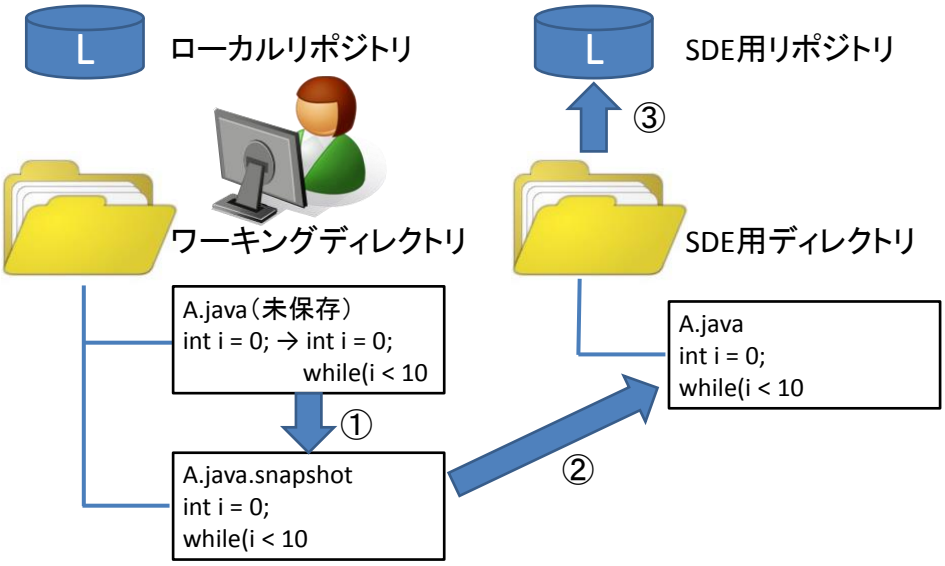

図 12: ファイル未保存時のプラグインの処理

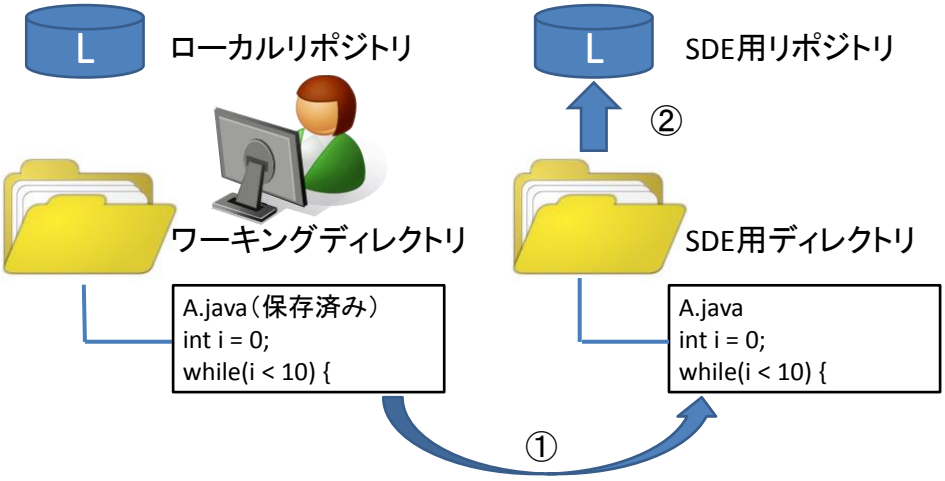

図 13: ファイル保存時のプラグインの処理

クアップする (図 12 ①)．次に，バックアップの処理に連動して，.snapshot ファイルの中身を取得し，SDE 用ディレクトリ内に作成した未保存状態のファイルと同名のファイルにコピーする (図 12 ②)．この処理によって，SDE 用ディレクトリ内にワーキングディレクトリの状態を再現することができる．そして，SDE 用ディレクトリの状態を SDE 用リポジトリにコミットする (図 12 ③)．

次に，Ctrl+S などのファイル保存動作に連動した，ファイル保存時の処理について説明する．プラグインは IDE 上で現在アクティブなファイルを対象とし，ファイルの中身を取得する．その上で，SDE 用ディレクトリ内に作成した対象ファイルと同名のファイルにコピーする (図 13 ①)．そして，SDE 用ディレクトリの状態を SDE 用リポジトリにコミットする (図 13 ②)．

以上の 2 つの処理を行うことでワーキングディレクトリの状態が細かい頻度で保存された細粒度作業履歴を SDE 用リポジトリに収集することが出来る．

4.2 行単位作業履歴作成ツールの実装

行単位履歴作成ツール (以下，作成ツール) 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出力とする．作成ツールは Java プログラムとして実装を行った．

4.2.1 開発環境

開発環境，ライブラリは上記で説明したプラグインと同様のものを用いた．加えて，ファイルの差分を扱うためのライブラリとして java-diff-utils[8] を用いた．

4.2.2 処理の流れ

細粒度リポジトリにアクセスし，各コミットに関する以下の情報を取得しリビジョン情報としてリスト化する．一つのリビジョンには一つのファイルに関する情報が記録されている．

- id
- ファイルパス
- ファイル内容
- 日付
- コメント
- 親コミット id

- コミット者

リスト化した各リビジョンについて，自分と同じファイルを扱い，かつ，自分よりも古いリビジョンを捜索する．見つけた古いリビジョンを親リビジョンとする．自分と親リビジョンとの差分が自分の changeset となる．差分は java-diff-utils から Patch 形式で得ることが出来る．Patch は隣接した変更の塊である Delta のリストで表される．Delta は変更前と変更後のファイル情報である 2 つの Chunk で構成されている．Chunk は変更前，あるいは変更後の文字列リスト，変更の開始行番号で構成される．そこで，Chunk 内の文字列リストが 1 行ずつ増えるような複数の Patch を作成し，それらを親リビジョンのファイル内容に適用することで，作業の推移を 1 行ずつ表す複数のファイルを出力する．出力したファイルを随時コミットすることで行単位作業履歴を作成する．

4.3 TLC 支援ツールの実装

TLC 支援ツールは行単位作業履歴を入力とし，TLC が実現されたリポジトリを出力とする．TLC 支援ツールは Scala により作成されたツールである “Uchronie” [5] に機能を追加する形で実装を行った．TLC 支援ツールは行単位リポジトリ内にブランチのコピーを生成し，そのブランチを操作し開発者が望む形に変えた上で，元のブランチを上書きする．3 章で説明した機能は MoveOp を除き全て Git 本来の機能として存在する．MoveOp に関しては別のファイルを扱うコミット同士の順序変更については Git の機能で実現可能である．しかし，このツールの利用目的から同じファイルを扱うコミット同士でも順序変更を行うことが出来るのが望ましい．そこで，Uchronie の MoveOp に同ファイルを扱うコミットの順序変更機能を追加した．図に機能の概要を示す．前提として，Git のコミットの順序変更はコミットの持つファイル内容ごと順序を変えるものである．同じファイルを扱うコミットの順序を上記の機能で変更すると，開発履歴の矛盾を引き起こすだけである．そのため，同じファイルを扱うコミット間の順序変更については，changeset の順序変更を行うように実装を行った．

順序変更を行うコミット A，B および A と B と同じファイルを扱い，A と B より一つ古いコミット Z を考える．変更前の順序は Z → A → B であり，これらのコミットを Z → B(変更後) → A(変更後) の順序に変更する．処理においてコミット X の持つファイル内容を F(X)，2 つのコミット X，Y 間の changeset を c(X, Y) と表す．この時，3 つのコミットが適用された時点で全ての変更がソースコードに存在する必要がある．そのため，最終的なファイルの内容は $F(B) = F(A(\text{変更後})) = F(Z) + c(Z, A) + c(A, B)$ とする必要がある．そこで，B(変更後) のファイル内容を，$F(B(\text{変更後})) = F(B) - c(A, B)$ とすることで B(変更後) は Z に c(Z, A) より先に c(A, B) を追加したものとなる．

4.4 利用フロー

本論文で提案したツールの利用フローについて説明する．利用フローは環境のセットアップ，開発作業中の細粒度作業履歴収集，行単位作業履歴作成ツールの利用，TLC 支援ツールの利用の 4 つに別れる．

それぞれの詳細を以下で説明する．前提として開発を行うファイル一式はリモートリポジトリ上に存在することとする．

4.4.1 環境のセットアップ

環境のセットアップ時のフローを図 14 に示す．

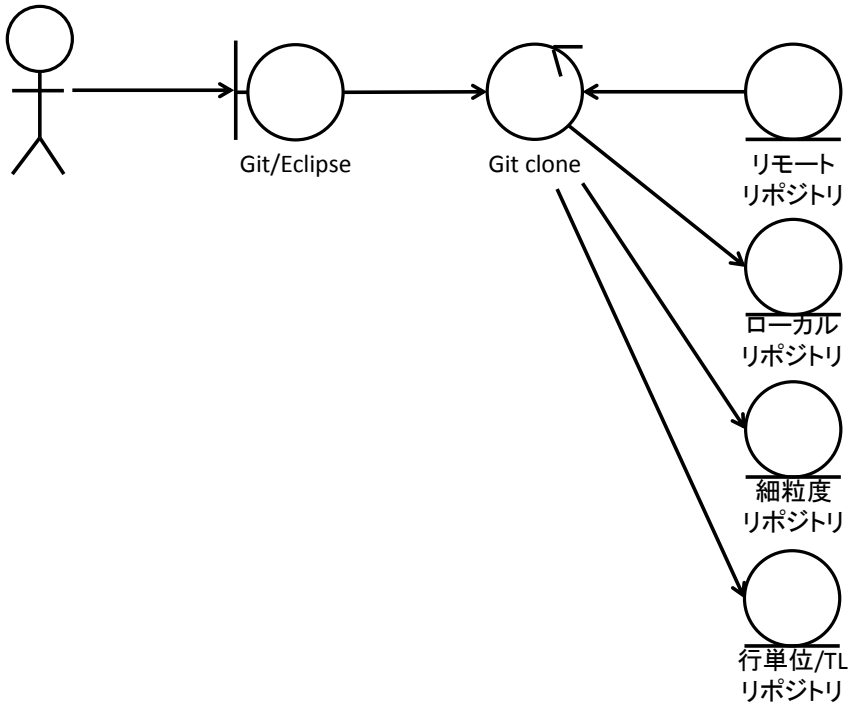

図 14: セットアップ時のフロー

ツールの利用を始めるにあたって，まず Git の設定を変更する．ソフトウェア開発において，グループで行う場合開発者によって開発環境が違う可能性がある．開発環境の違いから来るファイル形式の違いの一つとして改行コードがある．一般的に，改行コードは表 3 に表される 2 種類のコードのどちらか，もしくは組み合わせで表される．

現在使われている OS では表 4 の改行コードがデフォルトで扱われる．

改行コードが違うファイルは一見すると同じ内容に見えても，VCS 上では全く違う内容として扱われる．Git にはこのようなファイルの形式の違いを吸収する機能が存在する．“git config core.autocrlf xxxx” を設定することで，リポジトリとファイルをやり取りする際に改行コードをどう扱うかを設定することが出来る．上記の xxxx には true，false，input が入る．それぞれの設定を表 5 に示す．

本論文では Windows 上でのみツールを使うことを前提に，core.autocrlf を false に設定する．次に，図 14 にあるように，リモートリポジトリからローカルの 3 箇所にファイルをチェックアウトする．

以上で，セットアップが完了する．

4.4.2 開発作業中の細粒度作業履歴収集のフロー

開発作業中のフローを図 15 に示す．

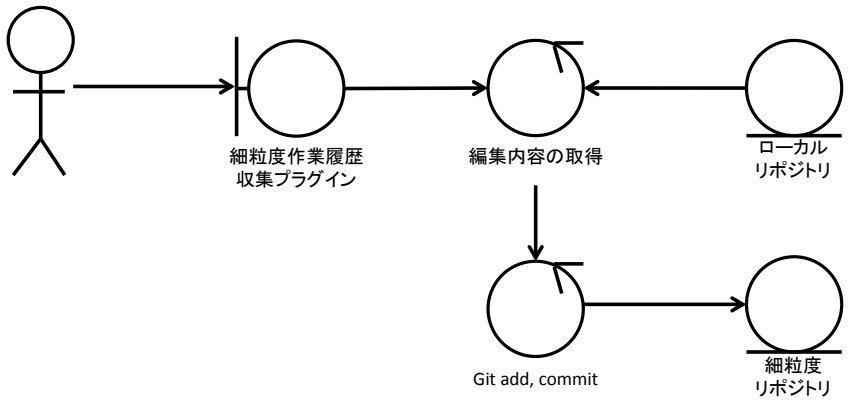

図 15: 開発作業中の細粒度作業履歴収集のフロー

開発作業中，開発者は特別な作業を行う必要は無い．IDE 上での作業は自動的に細粒度リポジトリに記録される．

表 3: 改行コード一覧

記号	呼び名, 別名	ASCII コード	マッチング
LF	Line Feed, New Line	0A	`\n`
CR	Carriage Return, Return	0D	`\r`

4.4.3 行単位作業履歴作成時のフロー

行単位作業履歴作成時のフローを図 16 に示す.

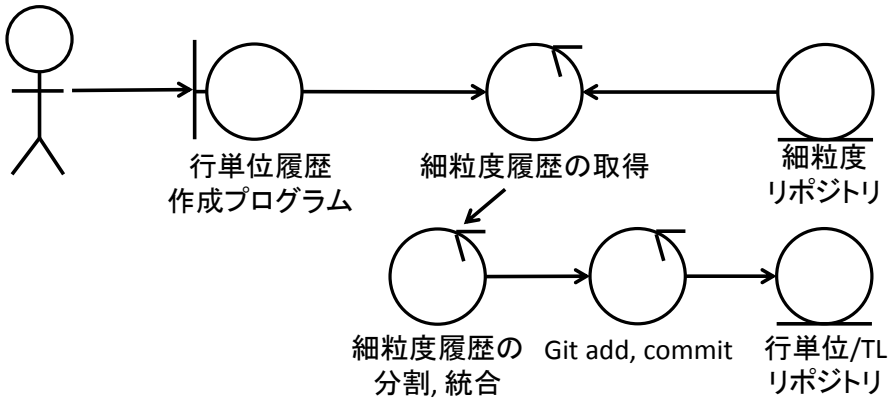

図 16: 行単位作業履歴作成時のフロー

行単位履歴作成ツールは Jar 形式のツールとなっており，引数に “細粒度リポジトリの絶対パス” と “行単位リポジトリの絶対パス” を与えることで，細粒度リポジトリ内のコミットを読み込み，行単位リポジトリに行単位作業履歴をコミットする.

4.4.4 TLC 支援ツール利用時のフロー

TLC 支援ツール利用時のフローを図 17 に示す.

TLC 支援ツールも行単位履歴作成ツールと同様に Jar 形式のツールとなっている．パスに “行単位リポジトリの絶対パス” を与えることで，行単位リポジトリから行単位作業履歴を取得し，一覧として表示する．TLC 支援ツールを用いて行単位作業履歴をタスク単位で統合することで TLC を実現する．最終的に，Git の機能を用いて TLC を実現したコミットをリモートリポジトリにプッシュすることで，自分以外の開発者に行った変更を公開することが出来る.

表 4: OS のデフォルト改行コード

改行コード	使用機種
LF	UNIX 系 OS
CR	マッキントッシュ
CR+LF	Windows(MS-DOS や WindowsNT も同様)

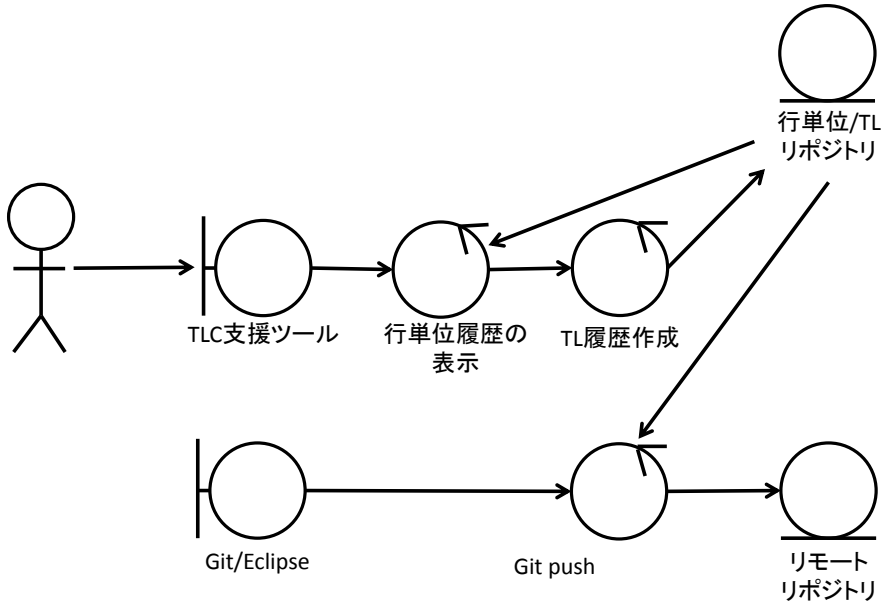

図 17: TLC 支援ツール利用時のフロー

5 評価実験

5.1 行単位作業履歴作成手法の妥当性評価

行単位作業履歴作成手法の正当性を評価するための実験を行った.

5.1.1 実験概要

過去に収集された作業履歴のサンプルデータに対し，行単位作業履歴作成ツールを適用し，正常に行単位作業履歴が出力されるかを評価した.

作業履歴は 2012 年度 IT Spiral[7] において行われたグループ開発で収集した 1 グループ, 6 人のものである．サンプルデータは Java で作成された約 6000 行規模のプログラムに対す

表 5: Git の改行コード変換設定

設定	コミット時	チェックアウト時
true	CRLF → LF	LF → CRLF
false	何もしない	何もしない
input	CRLF → LF	Windows ならリポジトリ内の改行コードを crlf に変換

る作業履歴である．

サンプルデータはオンラインストレージを用いた作業履歴収集手法により収集された [23]．オンラインストレージとはインターネット上でファイル保管用のディスクスペースを貸し出すサービスである．この手法はオンラインストレージ上でファイルを保存するたびにサーバー上に保存されるファイルの履歴を取得し，データベース保存する．本論文の手法とは違い，履歴が保存されるタイミングがファイルを保存した時のみであるため，本論文の手法に比べて履歴の粒度は少し大きくなっている．

5.1.2 評価基準

ツールが正常に行単位作業履歴を出力しているかを判断する際，行単位作業履歴の数が膨大になるため，目視では評価が難しい．そこで，プログラムを用いて行単位作業履歴として正常に作成されなかった履歴を捜索することで評価を行った．捜索する履歴の条件は以下である．

- 複数ファイルの changeset が含まれている履歴
- changeset がファイルの複数箇所に存在している履歴
- 複数行にまたがる changeset が存在している履歴

この条件を満たす履歴が存在していなければ，行単位作業履歴が正しく出力されていると判断する．

5.1.3 結果と考察

評価実験の結果，行単位作業履歴を正しく出力できることを確認した．ただし，目視での確認中に 1 つのパターンでソースコードへの変更順序を正しく再現できないことが判明した．そのパターンを図 18 に示す．

本手法では，細粒度作業履歴の changeset を行単位で分割し，上の行から順に履歴に追加していく．そのため，細粒度作業履歴の changeset 内で下から上に変更が行われた場合，間違った順序で行単位作業履歴を追加してしまう．細粒度作業履歴は数秒単位で収集するため，手動での変更であればこのパターンが発生することはほとんど無い．しかし，図にあるようなコードの場合，開発者が for 文などを記述した直後に，IDE によって自動的に閉じ括弧 (}) が生成され，手作業で括弧の中を記述するという状況が発生する．

このパターンにおいて変更の順番を正しく取得するためには，IDE によるコードの自動生成を検知し，自動生成されたコードのみを纏める機能があれば良いと考える．

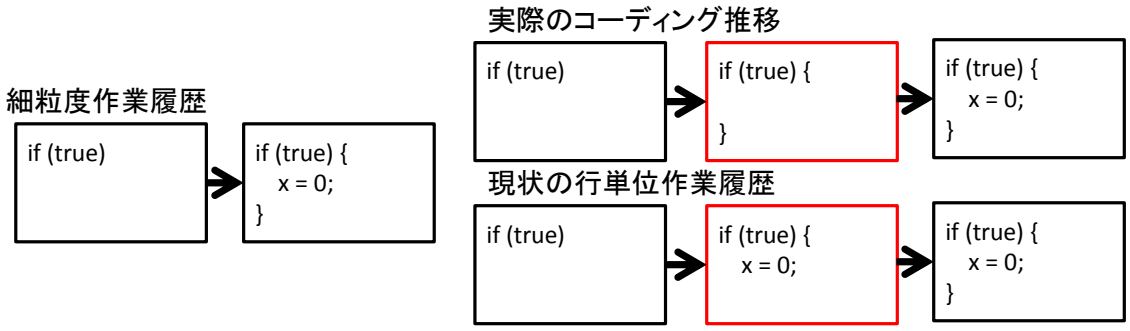

図 18: ソースコードへの変更順序を再現できないパターン

5.2 Task Level Commit 支援の有効性評価

本論文で提案したツールを用いることで，一般的に用いられている VCS を用いた開発環境と比べてどれだけより TLC を順守することが出来るかを評価した．

5.2.1 実験概要

本論文で提案したツールと既存の Git 公式ツールを比較を行った．本実験では，被験者にソフトウェアの開発を行なってもらい，成果物を Git 公式ツールか，提案ツールの一方を用いてコミットを行なってもらった．

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．提案ツールに関しては，セットアップの方法と TLC 支援ツールの利用方法を説明した．また，公式ツールについては基本的なコミット方法のみを指示した．

実験では，被験者にソフトウェアの仕様書と開発する際のタスクリストを与えた．被験者には仕様書に従ってソフトウェアを開発し，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．コミットの際はコーディングの前半と後半で提案ツール，Git 公式ツールのうち指定したものを利用してもらった．また，出来る限り TLC を順守したコミットを行うよう心がけ，タスクリストに記載されたタスク以外の作業が発生した場合は，独立したタスクとしてコミットするように指示した．

被験者は大阪大学大学院 情報科学研究科の大学院生 4 人に協力をお願いした．被験者は全員が Java プログラミングの経験者であり，Git の利用経験はなかったため，コーディングやツールを利用する際のスキルに大きな違いはないと考えられる．

開発するソフトウェアは Java で作成されたオセロゲーム [9] に実験用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．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．オセロゲームの完成時の GUI を図に示す．

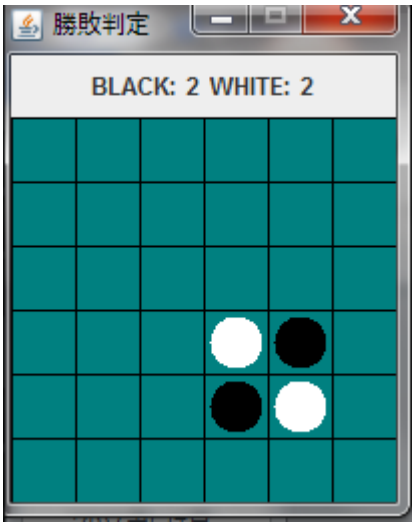

図 19: オセロゲーム

実験に課題の難易度やツールの利用順が影響を与える可能性があるため，4 人の被験者はそれぞれ実行するタスクリストと使用するツールの順番を以下の形で偏りがないように設定した．

- 前半タスクリストを先に実行，提案ツールを先に使用
- 後半タスクリストを先に実行，提案ツールを先に使用
- 前半タスクリストを先に実行，公式ツールを先に使用
- 後半タスクリストを先に実行，公式ツールを先に使用

5.2.2 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，作業時間とツールの使用感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 $* 100(\%)$
という式で定義した.

作業時間は細粒度作業履歴が記録され始めた時間から，TLC リポジトリが作成された時間までに設定した.

ツールの使用感は各ツール利用後に被験者に回答してもらったのアンケート結果に基づいて比較した．インタフェースの満足度を質問紙ベースで行う手法の一つに SUS(System Usability Scale)[14] と呼ばれる手法がある．SUS において被験者は特定のユーザインタフェースを対象として，有効性，効率性，満足度といった 10 項目それぞれについて 1(まったくそう思わない)〜5(まったくそう思う) の 5 段階で評価する．その後，被験者の回答結果を総合した SUS スコアを算出することで，特定の被験者が異なるインタフェースを利用した時にどのように評価が変わったかを相対的に知ることが出来る．以下に 10 の質問項目の内容を示す.

(1) このシステムをしばしば使いたいと思う

(2) このシステムは不必要なほど複雑であると感じた

(3) このシステムは容易に使えると思った

(4) このシステムを使うのに技術専門家のサポートを必要とするかもしれない

(5) このシステムにあるさまざまな機能がよくまとまっていると感じた

(6) このシステムでは，一貫性のないところが多くあったと思った

(7) たいていのユーザーは，このシステムの使用方法について素早く学べるだろう

(8) このシステムはとても扱いにくいと思った

(9) このシステムを使うのに自信があると感じた

(10) このシステムを使いはじめる前に多くのことを学ぶ必要があった

上記の指標のうち，奇数番号の項目はポジティブな指標，偶数番号の項目はネガティブな指標となっている．そのため，奇数番号の指標は回答から 1 を引いた値，偶数番号の指標は 5 から回答を引いた値として 0〜4 の値に変換し，これらの値の合計に 2.5 をかけた値を評価値として利用する.

また，ツールの利用感については被験者に対し，口頭での感想の聞き取りも行った.

5.2.3 結果

TLC 順守率に関する結果を表 6 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．以上から，提案ツールの TLC 順守率が公式ツールを上回る結果となった．

次に作業時間に関する結果を表 7 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 分となり，提案ツールを利用した場合のほうが作業時間が長いという結果となった．

最後に，アンケートの結果を示す．表 8 にツールの利用感を表す SUS スコアの算出結果を示す．被験者全員が公式ツールをより高く評価する結果となった．

また，被験者に聞き取りでアンケートを取った結果得られた感想を以下に示す．

- タスクを気にせずひたすらコードを書いてからコミット出来るのが良かった
- 履歴とタスクの結びつけをゲーム感覚で出来てよかった
- ツールの使い方は難しくなかった

表 6: TLC 順守率

被験者	提案ツール (%)	公式ツール (%)
A	100	66.7
B	100	83.3
C	66.7	50
D	83.3	66.7
平均	87.5	66.7

表 7: 作業時間

被験者	提案ツール (TCL 順守支援ツールのみ)(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

- タスクの境界がはっきりしている開発に有効だと思う
- 行単位の履歴は粒度が細かすぎて作業を思い出せない
- 開発時の環境でコードと比較しながら行単位作業履歴を纏めたい
- 行単位作業履歴の数が多すぎる

5.2.4 考察

まず，TLC 順守率に関して考察を行う．本実験では提案ツールを用いたほうが TLC 順守率が高いという結果となった．これは行単位作業履歴において変更された行とタスクの関連さえ理解できれば，あとは 2 つを関連づけることで TLC を順守できるという提案ツールの強みが出た結果と考えられる．

まず，被験者に聞き取り調査を行った結果，公式ツールで TLC が順守できなかった理由は以下の 2 つであった．

- 仕様書に従いコードを書いていたら次のタスクまで開発を始めてしまっていた
- 機能を全て実装し終わる前にコミットしてしまった

この 2 つが原因となる TLC が順守出来なかった事例は提案ツールを利用した場合では発生しなかった．

提案ツールで TLC が順守できていない原因は，変更とタスクの関連が分からず結びつけができなかったことである．提案ツール上で changeset を表示する際，周囲の数行も同時に表示される．図に changeset の表示例を示す．

表 8: SUS

被験者	ツール	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	評価値
A	提案ツール	1	1	1	4	2	3	3	1	2	3	52.5
	公式ツール	3	3	2	1	2	4	2	3	1	2	57.5
B	提案ツール	1	4	1	3	3	1	1	1	2	2	47.5
	公式ツール	2	3	2	2	2	2	3	3	1	2	55
C	提案ツール	3	1	2	4	0	3	4	1	3	3	60
	公式ツール	2	3	4	4	2	2	4	3	1	1	65
D	提案ツール	3	2	2	4	1	4	4	3	3	3	72.5
	公式ツール	3	3	4	4	3	3	4	2	3	3	80

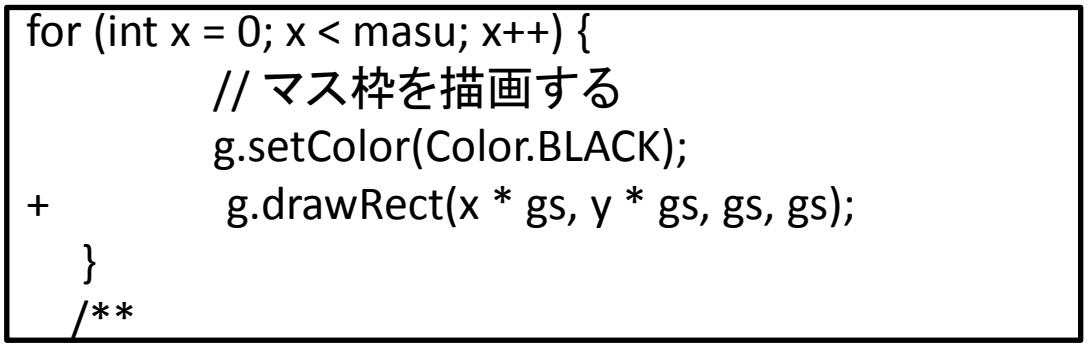

図 20: changeset の表示例

行の先頭に “+” が記述されている行が追加が行われた changeset である．このような表示では被験者は changeset が何のタスクに結び付けられるべきか理解できなかった．その結果，被験者はこの changeset を “タスクの不明な変更” として一つのコミットに纏め，他のタスクにコードの不足が発生した．

次に作業時間に関する考察を行う．本実験では提案ツールを使うことで作業時間が伸びる結果となった．その原因として考えられるのが TLC 支援ツールの利用時間の長さである．図 7 にあるように，提案ツールの作業時間は TLC 支援ツールを利用する時間が半分を占めた．そのため，純粋なコーディングの時間は 36.7 分程度だったと考えられる．公式ツールを使用した場合のコーディング時間が同じ長さだった場合，公式ツールでコミットを作業を 6 回行う時間が 62.6-36.7 = 25.9 分と仮定できる．つまりコミット 1 回が 4.3 分という計算となる．“git add .”，“git commit -m ”コミットメッセージ”” という 2 つのコマンドで実行できる公式ツールのコミットに 1 回 4.3 分かかるとは考えづらい．そのため，コーディング時間は公式ツールを利用した場合より提案ツールを利用した場合の方が短かったと考えられる．

次にツールの利用感のアンケートに関する考察を行う．SUS による評価で公式ツールの方が高く評価された原因の一つは，公式ツールで利用できる機能を制限していたことが理由だと考えられる．3 章で説明したように，TLC 支援ツールの機能は公式ツールでも実現可能であるが，知識や経験が必要なものとなっている．一方で，この評価実験では公式ツールの機能を単純なコミットのみに制限した．そのため公式ツールで利用する機能が提案ツールより少なくなり，利用する際の難易度が大幅に下がった．結果，提案ツールに比べ，公式ツールの使いやすさの評価がより高まった可能性がある．

聞き取り調査の結果では提案ツールの利用する際の難易度はそれほど高くなかったと推測される．また，提案ツールを利用することでタスクを気にせずひたすらコードを書くことができるという点がコーディング作業が短縮される理由であると考えられる．一方で，行単位

作業履歴の粒度の細かさが提案ツールの利用感に悪影響を与えてることも聞き取り調査で判明した．行単位作業履歴の数が多い点も，細かすぎる点も，より粒度が荒い履歴を提示することで解決すると考えられる．ただし，提案ツールでは手動での履歴の分割ができないため，履歴の粒度を荒くしてもその中に複数のタスクが存在しないことを保証する必要がある．

6　関連研究

ChEOPS[17] や SpyWare[30] をはじめ, 開発履歴の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.

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 [19][28][29]. これらは履歴の再利用などには対応していない.

本研究と同様にソースコードの編集履歴を収集し，編集履歴の理解性や利用性を向上させるために, 編集履歴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研究を林らが行なっている [34]．Historef は編集履歴の 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ている. 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ている.

主な違いとしては，林らの研究は一定のパターンに従い自動で編集履歴をリファクタリングしている一方で，本研究は手動である代わりに開発者の任意の履歴を構築する手段を提供している．また，本研究では履歴を Git を用いて扱っているため，独自のツールで扱う Historef よりも再利用性が高い.

7　あとがき

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，手作業による行単位作業履歴のタスクレベルでの統合支援をキーアイディアとした手法を提案した．

また実際に，細粒度作業履歴収集プラグイン，行単位作業履歴作成ツール，TLC 支援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．

また，評価実験として大学院生 4 人を対象に評価実験を行った．結果として，TLC 順守率は Git の公式ツールを利用した場合の 66.7%に比べ，87.5%と高い確率となった．

今後の課題として，IDE によるコードの自動補完の検知によるコード変更順序の修正，行単位作業履歴の粗粒度化，フィルタリング機能の実装，および GUI の充実を目指していきたい．

謝辞

謝辞本研究において，貴重な時間を頂いて懇切丁寧なご指導及びご助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より感謝いたします．大学院在籍の 2 年間，本当にお世話になりました．

研究室での作業中に声をかけて下さったことや，私の研究にピンポイントな関連研究を紹介してくださるなど，大学院生活での励みを頂きました．定年も近いとのことですが，これからの一層のご活躍を期待しております．

本研究において，適時適切なご指導及びご助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします．

研究グループの取りまとめなど，大学院における快適な研究生活は松下先生のおかげだと思っております．また，研究室内での発表で議論が発生した時，議論の内容を纏めてくださるのも松下准教授でした．議論についていけなくなることも多い中，非常に助かりました．またラーメン屋で偶然出会えることを楽しみにしております．

本研究において，日常の議論を通じて多くのご指導及びご助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く感謝いたします．

評価実験のプランニングや，研究に関連する細かい知識など，石尾先生には様々なことを教わりました．石尾先生の行った評価実験に比べると粗ばかりが目だつ実験となってしまいましたが，それでも結果を出せたのは石尾先生のおかげです．

本研究において，終始丁寧に適切なご指導及びご助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井垣宏特任准教授に深く感謝いたします．

楠本研究室での卒論執筆から 3 年間，直接ご指導していただき，本当にお世話になりました．研究内容だけでなく生活態度に至るまで，一から十までご迷惑ばかりおかけして，申し訳ない気持ちでいっぱいです．それでも対外発表のたびに「前より良くなった」と言ってもらえた事が，様々なプレッシャーに対する励みになりました．改めて，本当にありがとうございました．

本研究において，随時適切なご指導及びご助言を頂きました奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学講座吉田則裕助教に心より感謝いたします．

本論文で紹介したコミットポリシーや，いくつかの論文などをそっと提供してくださり，何度助けられたかわかりません．そしてそれ以上に，突然の結婚報告で研究室周辺を狂乱の渦に巻き込んだあの日を私は忘れません．私も吉田先生のように周囲をあっと驚かす事が出来る人間になりたいと思います．

本研究において，多くのご指導及びご助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻崔 恩瀞氏に深く感謝いたします．

発表練習のたびに一番多くのアドバイスや添削を下さったのが崔さんでした．それだけでなく，日常生活でも何かと気を使ってくださり非常に助かりました．吉田先生と末永くお幸せになられることをお祈りしています．ありがとうございました．

本論文の執筆において，文章構成や執筆方法に関して様々なご指導及びご助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻堀田圭佑氏に心より感謝いたします．

堀田先輩の論文添削はユーモアと優しさにあふれていて，修論が大詰めで荒んでいた私の心を癒してくれました．堀田先輩のような後輩への指導が出来るよう精進していきます．

本研究において，突発的な評価実験に付き合ってくださった大阪大学大学院情報科学研究科情報ネットワーク専攻通阪航氏に深く感謝いたします．

私の家に泊まりに来た貴方に対し，“宿泊代だ，評価実験をやるぞ” と言い放った私に 3 時間近く付き合ってくださり本当にありがとうございました．私は来年度から東京に住むことになりますが，貴方もおそらく定期的にこちらに来るのでしょう．これからもちょくちょく遊びましょう．

最後に，その他様々なご指導，ご助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室，楠本研究室の皆様に深く感謝いたします．

参考文献

[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] Darcs. http://darcs.net/.

[4] Git. http://git-scm.com/.

[5] 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).

[6] Gitblit. http://gitblit.com/.

[7] It スパイラル. http://it-spiral.ist.osaka-u.ac.jp/.

[8] java-diff-utils. http://code.google.com/p/java-diff-utils/.

[9] 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.

[10] Jgit - eclipse. http://www.eclipse.org/jgit/.

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

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

[13] Christopher Alexander. The timeless way of building. 1979.

[14] Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, Vol. 24, No. 6, pp. 574–594, 2008.

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

[16] Marco D'Ambros, Michele Lanza, and Romain Robbes. Commit 2.0. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, pp. 14–19. ACM, 2010.

[17] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pp. 3–24. ACM, 2007.

[18] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[19] Lile Hattori, Mircea Lungu, and Michele Lanza. Replaying past changes in multi-developer projects. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pp. 13–22. ACM, 2010.

[20] 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.

[21] 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.

[22] 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.

[23] Umekawa Kohichi, Hiroshi Igaki, Yoshiki Higo, and Shinji Kusumoto. A study of student experience metrics for software development pbl. In *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pp. 465–469. IEEE, 2012.

[24] Martin Lippert and Stephen Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully.* John Wiley & Sons, 2006.

[25] 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.

[26] 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.

[27] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 31–34. ACM, 2008.

[28] Takayuki Omori and Katsuhisa Maruyama. An editing-operation replayer with highlights supporting investigation of program modifications. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pp. 101–105. ACM, 2011.

[29] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, Vol. 166, pp. 93–109, 2007.

[30] Romain Robbes and Michele Lanza. Towards change-aware development tools. Technical report, Technical Report 6, Faculty of Informatics, Università della Svizzerra Italiana, Lugano, Switzerland, 2007.

[31] Muneo Takahashi. Software configuration management and a tendency of its standardization. *IPSJ SIG Notes*, Vol. 96, No. 71, pp. 39–46, jul 1996.

[32] 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.

[33] 藤原克則. *Nyūmon Mercurial: Linux Windows taiō*. 秀和システム, 2009.

[34] 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.