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.

コーディングにおける細粒度作業履歴を用いた手戻り支援ツールの検討

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　コーディング中に作業を遡りソースコードを以前の状態に戻す作業を手戻りという．手戻りは故障 (failure) が発生し，バグがどこにあるか分からない場合などに，プログラムを正常に動く状態に戻すためによく行われる．
手戻りを支援するツールとしては Subversion 等のバージョン管理システム (VCS) や，Eclipse などの IDE においてソースコードに加えた編集作業を記録し，改訂履歴を提供するローカルヒストリー機能が存在する．しかしながら，VCS で記録される変更履歴はバグ修正時の手戻りを行うには粒度が荒すぎることが多い．また，ローカルヒストリー機能では作業履歴が自動で保存されるため，コードの構成要素を分断する形で作業履歴が保存される場合がある．結果として，プロジェクト全体をバグを混入させた直前の状態に戻すことが困難になる．
そこで本研究では，一定時間ごとにプロジェクトの状態を自動でバージョン管理システムにコミットし，細粒度作業履歴を保存するツールを提案する．また，それによって作成された作業履歴を状況によって分割，結合することによって，細粒度リポジトリを作成する．
評価実験として，コーディング中にツールを適用した結果，指定したコードの構成要素で構築された細粒度リポジトリが構築できたことを確認した．

キーワード　ソフトウェア開発，手戻り，バージョン管理システム，IDE

Using fine grain version data to support software development rework.

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　Rework in coding process is usually conducted for reverting source code back to the old state in case of inducing bugs accidentally. There are some tools like Version Control Systems(VCS) and local history provided by Eclipse to support such rework. On the other hand, most of such conventional tools have granularity problems. For example, a time interval between changes of source code recorded in VCS is large relatively to revert the source code back to old state. On the contrary, local history may record source code too frequently. In this research, we propose a tool to make a fine-grained software repository automatically through developers' coding process. In our case study, we confirmed our tool could make a fine-grained repository which includes only one specific program element such as a line and a sentence in one revision.

Key words　Software Development, Software Development Rework, Version Control System, IDE

1. はじめに

ソフトウェア開発において，ソースコードは様々な理由により常に改修され続ける．その改修の過程において，正常に動作していたソフトウェアが異常な振る舞いを起こすようになることも多い．

このような異常な振る舞いを引き起こす原因（バグ）が容易に特定できるものであれば，そのバグを修正することで対応が可能である．しかしながら特定が困難なバグの場合，やむを得ずソースコードを正常に動作していた状態に戻すことがある．ここではソースコードを以前の状態に戻すことをソースコードの手戻りと呼ぶ．

ソースコードに対する手戻りは開発者によって手動で行われることもあるが，開発が進むにつれてファイルやファイルに含まれるコード量が膨大になると，手動ではどの状態にソースコードを戻せば良いかわからなくなる．また，手動での手戻りはそれ自体がバグの混入原因となる恐れもある．

そのため，通常この種の手戻りはバージョン管理システム [6] 開発環境（IDE）の機能を用いて行われることが多い．バージョン管理システムを用いることで，過去に開発者が保存した任意の状態にソースコードを手戻りさせることが可能となる．また一部の IDE には，開発者がコーディング中のソースコードを一定時間ごとやユーザが保存するごとに，手戻り可能な状態で保存する機能がある [8]．

一方でバージョン管理システムを用いた手戻りでは，開発者が過去に保存した状態にしか戻れないため，バグが混入されたと想定される箇所よりも相当前の段階に戻らざるをえない（過度の手戻りと呼ぶ）ことがある．また，IDE の自動保存機能では，ソースコードが文の途中等の不完全な状態で保存されていることも多いため，結果として手戻りが困難になることがある．

そこで我々は，これらの問題を改善し，開発者の意図した手戻り箇所に容易に戻れるようにすることを目的として，IDE の機能を利用した細粒度コーディング履歴を改行単位や文単位といった開発者の意図した手戻り単位の履歴に再構築し，バージョン管理システムに記録する手法を提案する．

以降，2 章では手戻りとそれを支援するツール，手戻りにおける問題点を説明する．また，3 章では手戻りにおける問題点を解決するためのキーアイディア，4 章ではキーアイディアを実現するための実装について説明する．5 章ではケーススタディとして実装したツールを既存の開発履歴に適用した結果について説明する．最後に 6 章ではまとめと今後の課題について述べる．

2. 準　　備

2.1 コーディングにおける手戻り

本稿において，コーディングにおける手戻りとはコーディング時にソースコードの記述を以前の状態に戻すことを指す．通常，正常に動作していたソフトウェアに対して修正や機能追加を行った際に故障 (failure) が発生し，その原因となるバグがどこにあるか分からない場合などに，プログラムを正常に動く状態に戻すために行われることが多い．手戻りの方法は様々で，正常に動いていた状態を確かめるために段階的にソースコードを戻したり，正常動作が確認されている初期状態まで一気に戻したりといったことが開発中に行われる．このような手戻りは手動で行われる場合もあるが，通常は既存のバージョン管理システムやそれに類するツールを利用して行われる．

バージョン管理システムとは，ソフトウェア開発の際に作成されるソースコードなどのファイル変更履歴を管理するシステムである [5]．開発者の操作 (コミット) によって開発中のソースコードにリビジョン番号が付与され，バージョン管理システム内のデータベース (リポジトリと呼ぶ) に保存される．開発者はリビジョン番号を指定することで，バージョン管理システムより過去の状態のソースコードを取得することが可能である．現在，Subversion [3] や Git [1] 等，非常に多くのバージョン管理システムが実装されている．

これらのバージョン管理システムではコミットという操作によって，開発者が意図したタイミングでソースコードを保存することができる．コミット時にはコメントの付与も可能となっており，バージョン管理システムにおいて良く実装されている現在のソースコードと過去の任意のリビジョンとの差分表示機能も併用することで，手戻り箇所の理解も容易であると考えられる．

また，ソフトウェアの統合開発環境（IDE）には，開発者が実装中のファイルを一定期間ごとに自動保存する機能を持っているものがある．例えば IDE の一つである eclipse にはローカルヒストリーと呼ばれる機能がある．ローカルヒストリーは，開発者が実装中のファイルを対象として新規作成や保存といった操作を行った際に，操作時のファイルを手戻り可能な状態で保存しておく機能である．ローカルヒストリーの機能を利用することで，開発者は IDE によって保存された過去の状態に手戻りすることが可能となる．

より細かい粒度でコーディング履歴を記録するツールとしては，Operation Recorder [7] が存在する．このツールは開発者が IDE のエディタ上で行った編集操作を編集履歴としてすべて記録し，その情報をデータベースに格納することで，プログラム変更理解のためのツール構築を支援する Eclipse プラグインである．編集履歴には，コードの編集やコピー&ペースト，リファクタリング機能の利用などが含まれる，Operation Recorder を用いることで，特定のソースコードに対して過去に行われた編集操作を，その種類や時刻に応じて容易に取得可能となるため，過去に行われた変更の意図を理解することが容易になると考えられる．

2.2 手戻りにおける課題

バージョン管理システムや IDE の自動保存機能を利用した手戻りでは，保存されるリビジョン間の時間間隔やソースコードの状態が問題となることがある．バージョン管理システムの場合，リビジョン間の時間間隔は開発者のコミットタイミングに依存する．そのためリビジョン間に開発されたコード量が非常に多くなることがある．そのため，開発者が手戻りを行いたいとおもったときに，間隔が長すぎて都合の良いタイミングに

戻れるリビジョンが無い場合がある．そのようなケースでは結果として手戻りができないか，戻れたとしてもバグが混入した可能性の高いタイミングを通り過ぎた過去の状態にしか手戻りができないことになる．

また，IDE のローカルヒストリーや OperationRecorder のような自動保存機能を用いた手戻りでは，バージョン管理システムと比較して時間間隔がより細かく記録されるため，開発者の意図したタイミングへの手戻りが行える可能性が高い．しかしながら，自動保存機能が実行されるタイミングは，開発者の意図しないタイミングで実施されるため，実装途中で保存された結果，行の途中までや文の途中までといったソースコードを構成する要素が分断された状態に手戻りが行われる可能性がある．結果として，手戻りを行った後のソースコードの理解が困難になり，その状態が正常であるかを把握するのが困難になることも考えられる．

以上より，手戻り操作においては，リポジトリに保存されているソースコードが下記要求を満たすことが望ましいと考えられる．

R1:　リビジョン間の時間間隔は十分短いこと

R2:　各リビジョンのソースコードにおいて，ソースコードを構成する要素が過度に分断されていないこと

R1 は開発者が戻りたい箇所どこにでも戻れるように，リビジョン間の時間間隔が短いことを示す．一方で，時間間隔が短すぎると，R2 で述べたソースコードの分断が発生する．そのため，十分に短い時間間隔でリビジョンが保存されており，各リビジョンに含まれるソースコードの状態が行単位，文単位，ブロック単位といった開発者の意図した最小構成要素を分断しない粒度であることが望ましい．そこで我々は IDE の自動保存機能を利用しつつ，リビジョンごとにソースコードの過度な分断が起きないようなリポジトリを構成する手法について次節で詳述する．

3.　細粒度作業履歴の作成

本章では,2.2 章で提示した既存の手戻り支援ツールが抱える問題に対する解決案を提示する．

3.1　キーアイディア

手戻り支援ツールの問題点を解決するためのキーアイディアを以下に提示する．

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と合成による細粒度リポジトリの作成

以下で，これらのキーアイディアの目的と，これらを実現するための自動的かつ定期的に細粒度作業履歴を収集する手法と，収集した作業履歴の分割と合成によって細粒度リポジトリを作成する手法を提案する．

3.2　全　体　像

キーアイディアの実現を行ったシステムの全体像を図 1 に示す．以降で,K1 を実現するための IDE による細粒度作業履歴の自動収集手法と, K2 を実現するための細粒度作業履歴の分割，合成に関する説明を行う．

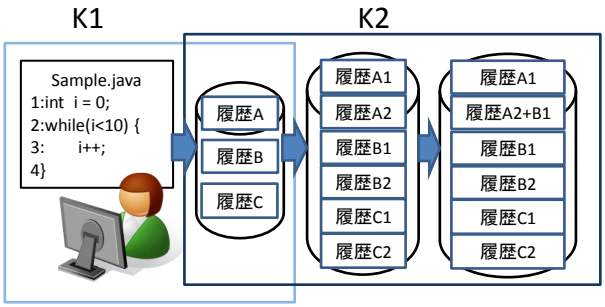

図 1　システムの全体像

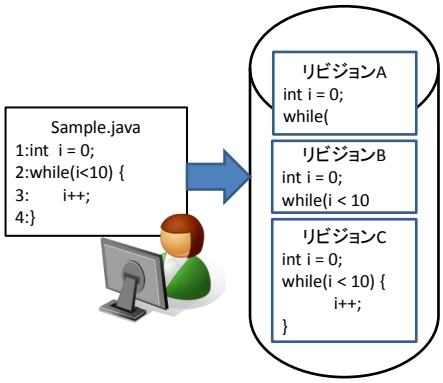

図 2　K1:IDE による細粒度作業履歴の自動収集

3.3　細粒度作業履歴の自動収集

2.2 でも説明した通り，バージョン管理システムはリビジョン間の時間間隔が長く都合の良い手戻り地点へ戻れない場合があり，また，IDE を用いた手戻りは文や行などのソースコードの構成要素の途中に手戻りが行われる可能性がある問題がある．そのため，開発者が意図したタイミングで，かつソースコードの構成要素を分断しない手戻りを行う手段が必要となる．そこで，本手法では 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．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．ここで作成される細粒度作業履歴は，ファイルごとに収集している．例を図 2 に示す．

3.4　作業履歴の分割と合成

K1 で収集した細粒度作業履歴はリビジョン間の時間経過が短い．しかし，自動的に収集するため，一部がコードの構成要素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，コードの構成要素を分断しない形に整形する．コードの構成要素とは，具体的には行単位，ある

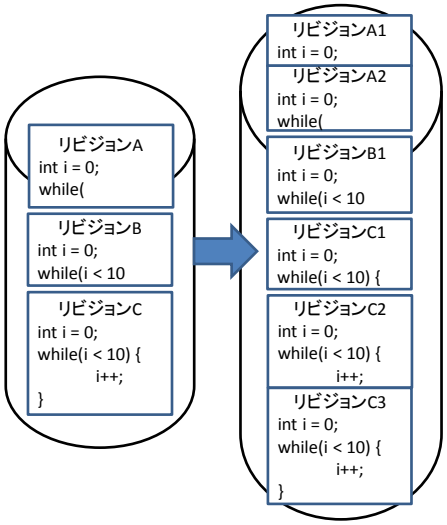

図 3　K2:収集した作業履歴の分割

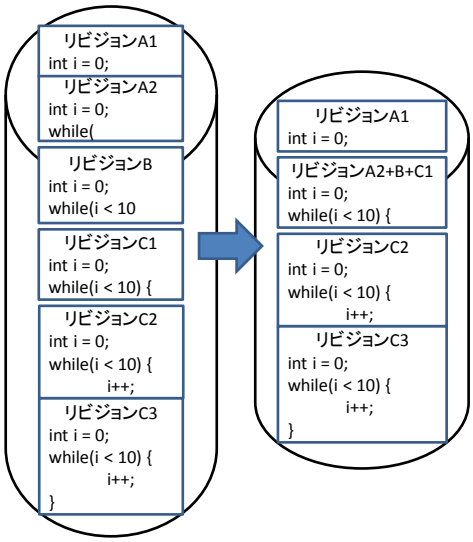

図 4　K2:同コード単位への作業履歴の合成

いはセンテンスなどの単位である．本論文では行単位での分割を行う．K1 で作成された細粒度作業履歴が持つ前回の作業履歴との差分を指定したコード単位で分割する．具体例を図 3 に示す．分割された細粒度作業履歴で構成された新しい細粒度作業履歴のリポジトリを作成する．K2 によって作成される細粒度リポジトリ内の 1 つの細粒度作業履歴には必ず 1 ファイルに対する 1 つのコードの構成要素に対する変更のみが含まれる．

細粒度作業履歴の分割によって作成された分割された細粒度作業履歴中では，同じコード単位に対する変更が 2 つ以上の作業履歴に分かれている場合がある．これは，K1 において自動で細粒度作業履歴を保存しているために，1 つのコードの構成要素に対する変更の途中で保存が行われるためである．そのため，1 つのコードの構成要素における変更を 1 つの細粒度作業履歴として合成する．4 に具体例を示す．細粒度作業履歴の合成によって作成される新しい作業履歴は，合成前のものと同様に必ず 1 ファイルの 1 つのコードの構成要素に対する変更のみが含まれる．ただし，合成前のものと違い同じコード単位に対する連続して変更を行う作業履歴は存在しない．

4. 実　　装

本ツールの実装についてキーアイディア K1，2 に分けて解説する．

4.1　K1 の実装

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 [2] を拡張して作成した．また作業履歴の保存のためのバージョン管理システムとして Git を用いて開発した．この Save dirty Editor Eclipse Plugin 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バックアップする
- ファイルを保存した時にバックアップ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，プロジェクトの状態を 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．

4.2　K2 の実装

細粒度リポジトリツールは，Java ソースコードにより作成されている．現状では K1 の実装とは独立したツールとなっている．

4.2.1　細粒度作業履歴の分割の実装

今回の実装では細粒度作業履歴を行単位で分割する．K1 で作成した各細粒度作業履歴において，その前後におけるファイルの差分を取得する．その差分を 1 行ずつ適用したファイルを新しいリポジトリにコミットする．この時，コミットコメントとして，

- 対象のファイル名
- 変更が行われた行番号

を追加する．以上で，1 ファイルに対する 1 行の変更をもつ細粒度作業履歴を記録した新しいリポジトリを作成する．

4.2.2　細粒度作業履歴の合成の実装

細粒度作業履歴の合成の際は，4.2.1 の実装において追加したコミットコメントを確認することで，同じファイルの同じ行への変更を確認することができる．連続していない変更と連続した変更のうち最後の変更のみを取得し，新しいリポジトリへコミットすることで，最終的に 1 ファイルに対する 1 行の変更をもち，連続して同ファイルの同行への変更を行わない作業履歴を作成する．

4.3　利用時の流れ

4.3.1　細粒度作業履歴の収集

開発者は細粒度作業履歴収集プラグインの jar ファイルを利用する Eclipse に予め導入する．その状態でコーディングを開始することで，開発者が特別な設定をすることなく，作業履歴の収集が自動的に開始される．作業履歴の収集に用いる Git のリポジトリは，コーディングを行なっているファイルが含まれるプロジェクトの直下に自動的に作成される．

```
 1:public class FizzBuzz {
 2:  public static void main(String[] args){
 3:   for (int i = 1; i <= 100; i++) {
 4:    if (i % 3 == 0 && i % 5 == 0)
 5:     System.out.println("Fizz,Buzz");
 6:    else if (i % 3 == 0)
 7:     System.out.println("Fizz");
 8:    if (i % 5 == 0)
 9:     System.out.println("Buzz");
10:    else
11:     System.out.println(i);
12:   }
13:  }
14:}
```

図 5　対象コード

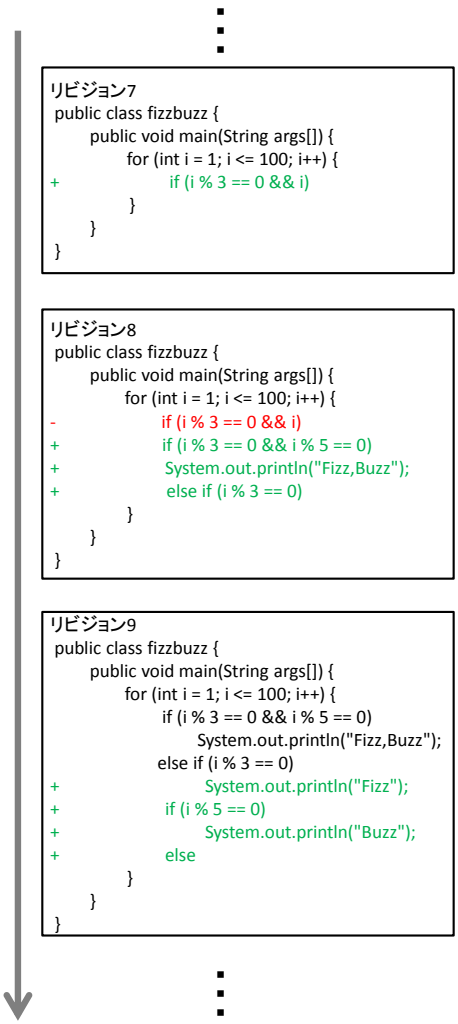

図 6　対象コードの細粒度作業履歴

4.3.2　細粒度作業履歴の閲覧，適用

現状，細粒度作業履歴の閲覧とプロジェクトへの適用は，Git が公式に配布している Git GUI を用いて行う．細粒度作業履歴を閲覧，適用したい場合，作業履歴収集プラグインで収集した作業履歴に対し，細粒度作業履歴作成ツールを適用し，作成された細粒度リポジトリを Git GUI を用いて閲覧，適用する．

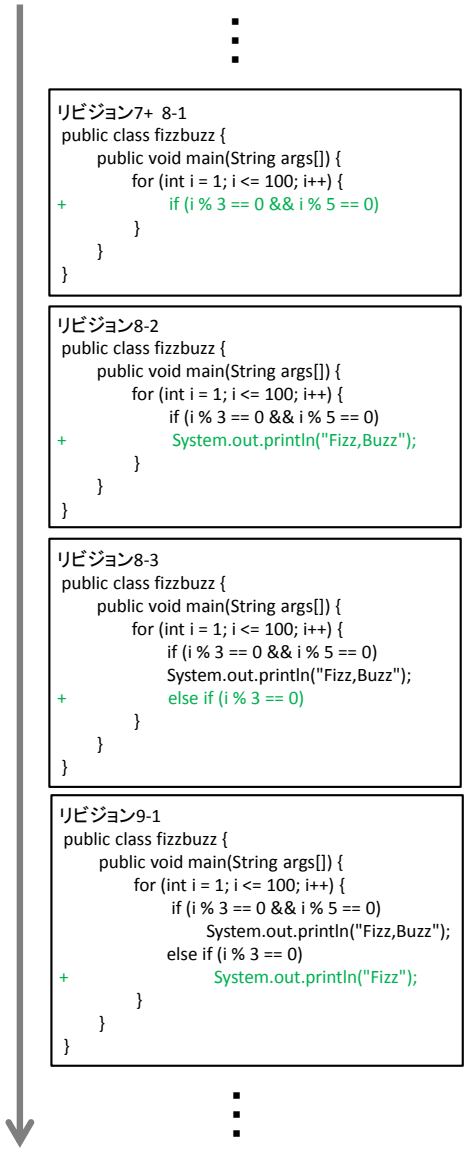

図 7　対象コードの細粒度リポジトリ

5.　ケーススタディ

ケーススタディとして，図 5 のコードを Eclipse 上で作成し，本研究で作成したツールを用いて細粒度リポジトリを作成した．そして，細粒度リポジトリが指定したコードの構成要素で構築されているかを調査した．なお，細粒度作業履歴を収集する際の，ファイルの変更から保存までの時間設定は 5 秒とした．また，コードの構成要素は行単位に指定した．

収集した細粒度作業履歴の一部を図 6 に示す．図の中で行の先頭に-がある赤い行は直前のリビジョンから削除された行，先頭に+がある緑の行は直前のリビジョンから追加された行である．ある行への編集は，削除した後に追加をしたと表記されるため，-の行の次に+の行がある場合，その行は編集が行われている．図 6 内では，リビジョン 8 の変更のうち最初の行は変更が行われている．

収集した細粒度作業履歴を本手法に従って分割，合成を行っ

た．以下，例として 10 個のリビジョンのうちリビジョン 7，8，9 に対する処理の説明を行う．まず，分割の処理について説明する．リビジョン 7 はリビジョン内に 1 行分の変更しか含まれていないため，分割の必要はなかったリビジョン 8 は削除が 1 行，追加が 3 行含まれていた．最初の削除 1 行と追加 1 行を合わせて編集 1 行と判断するため，このリビジョンは編集 1 行と追加 2 行でリビジョン 8-1 から 8-3 まで分割された．リビジョン 9 は，追加 4 行で 9-1 から 9-4 に分けられた．

次に同じ行への連続する変更を合成する処理について説明する．図 6 の中では．リビジョン 7 の追加とリビジョン 8-1 に対応する編集が同じ行への変更であった．そのためリビジョン 7 とリビジョン 8-1 は合成された．

図 7 に対し以上の分割と合成を行った細粒度リポジトリの一部が図 7 である．この細粒度リポジトリには 1 行単位の編集履歴が記録されており，このリポジトリを確認した結果，本研究で作成したツールは作業履歴を収集し，指定したコードの構成要素の単位で細粒度リポジトリを構築できることが確認できた．

6. おわりに

本研究では，コーディング過程における細粒度作業履歴を収集し，指定したコードの構成要素単位で構成されたリビジョンをもつ細粒度リポジトリを構築する手法を提案した．

今後は PDG を用いた Fault-localization 手法 [4] 等と組み合わせて，実際にバグが存在する可能性が高い箇所への手戻りを支援する仕組みを構築して行きたい．

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] Git.
[2] Save dirty editor eclipse plugin.
[3] Subversion.
[4] A. Askarunisa, T. Manju, and B. Giri Babu. Fault localization for java programs using probabilistic program dependence graph. *CoRR*, Vol. abs/1201.3985, , 2012.
[5] Peter H Feiler. *Configuration management models in commercial environments*, Vol. 258. Software Engineering Institute Pittsburgh, Pennsylvania, 1991.
[6] Mike Mason, でびあんぐる監訳. Subversion 実践入門:達人プログラマに学ぶバージョン管理 (第 2 版). オーム社, 2007.
[7] Omori Takayuki and Maruyama Katsuhisa. A method for extracting source code modifications from recorded editing operations. *IPSJ Journal*, Vol. 49, No. 7, pp. 2349–2359, jul 2008.
[8] 長瀬嘉秀ほか. Eclipse クックブック. O'Reilly Japan, 2004.