ソースコード編集履歴を用いたプログラム変更の検出

木津 栄二郎　大森 隆行　丸山 勝久

プログラム変更の検出のためには，そのプログラムの 2 つの版に対する行単位の差分を使用することが多い．しかしながら，単一の更新において複数の変更が存在している場合，それらを適切に解きほぐす必要があり，変更の理解者にとって手間がかかる．本手法では，検出対象のプログラムに対するすべての編集操作を記録した編集履歴を用いることで，ソースコードのスナップショットを復元し，2 つの連続するスナップショット間でそれぞれの構文解析の結果を比較する．これにより，従来より細かい時間間隔でのプログラム変更を検出することができる．さらに，実際のプログラム作成に対して本手法を適用した実験結果を示す．

To detect the changes of a program, the line-based difference between two versions of the program have been frequently used. In a single commitment performed on a version control system, however, multiple changes are usually intermingled. In this case, it is troublesome to untangle them. The proposed method can detect such individual changes. For this, it restores snapshots of the program from the history of editing operations for the target program and compares information on class members that results from syntax analysis for respective snapshots. Experimental results with a running tool implementing are also shown.

1　はじめに

ソフトウェア保守においては，保守対象であるプログラムをよく理解した上で修正を加える必要がある．このためには，現在のプログラムのスナップショットだけでなく，過去にどのような変更が行われてきたのかを把握することが重要である [3].

プログラム変更の把握を目的として，版管理システムに保存された変更前後の 2 つの版のソースコードから diff などのツールを用いて行ベースの版間差分を抽出し，その差分から変更情報を検出する手法が従来からある．しかし，このような手法では検出できる情報に限界があることが指摘されている [4].

通常，版管理システムにおいて，ソースコード更新のコミット (更新の保存) は，開発者の考える自由なタイミングで行うことができる．このため，開発者や保守者の考える単一の更新に必ずしも 1 つの変更だけが含まれるわけではない．このような場合，版間差分に基づく従来手法では，混在する変更から個々の変更を検出することは困難である．このため，プログラムの理解者 (主に保守者) は，混在する変更を解きほぐす必要がある．このような作業は，プログラム理解者にとって面倒であり，時間的コストが大きい．

本論文では，開発あるいは保守時のソースコードに対する編集操作の履歴を用いて，プログラム変更を検出する手法を提案する．具体的には，過去の編集操作をソースコードへ再適用していくことで，編集操作が実際に行われた時刻ごとのソースコードを復元する．さらに，復元されたすべてのソースコードに対して，構文解析可能かどうかを検査する．構文解析可能な場合，ソースコードからクラスのメンバ (フィールドやメソッド) に関する情報 (クラス情報と呼ぶ) を抽出

Detecting Program Changes based on the Edit History of Source Code.

Eijirou Kitsu, 立命館大学大学院理工学研究科, Graduate School of Science and Engineering, Ritsumeikan University.

Takayuki Omori and Katsuhisa Maruyama, 立命館大学情報理工学部, Department of Computer Science, Ritsumeikan University.

コンピュータソフトウェア, Vol.29, No.2 (2012), pp.168–173.
[研究論文 (レター)] 2011 年 8 月 29 日受付.

する．最終的に，復元された 2 つのソースコード間でクラス情報を比較してプログラム変更を検出する．

ここで，本研究における編集操作とは，開発者や保守者が編集中のソースコードに対して行ったすべてのテキスト書き換えを指す．本手法では，OperationRecorder [1] により，Eclipse のエディタ上で行われた Java ソースコードに関する編集操作が記録されていることを前提としている．また，本研究におけるプログラム変更とは，Java ソースコードを構文解析した結果として得られるプログラム構文要素に関する変化を指す．本論文では，Java プログラムに対する影響分析 [2] [5] において提唱されているプログラム変更の分類を参考に定義した．本手法により，従来の手法に比べて，より細かい時間間隔でプログラム変更が検出可能である．

本論文では，まず 2 章で，本研究で新たに検出可能となるプログラム変更の例題を示す．次に，3 章で，編集履歴を用いたプログラム変更検出手法について述べる．4 章で，提案手法を用いた予備実験の結果を示す．最後に，5 章でまとめと今後の課題を述べる．

2 例題

図 1 に示すソースコードを用いて，従来の手法では細かく検出できないプログラム変更の例を説明する．この図では，単一のソースコードに対して，以下の 3 つの変更が行われている様子を表している．

1. クラス A のメソッド X をクラス B に移動
2. クラス B のメソッド X の名前を Y に変更
3. クラス B のメソッド Y の本体を編集

いま，これら 3 つの変更が行われた後に更新がコミットされた場合を考える (1 番目の変更前を第 n 版とすると，第 3 番目の変更後が第 $(n+1)$ 版となる)．第 n 版と第 $(n+1)$ 版のソースコードから diff により差分情報を抽出すると，以下のようになる．

```
2,3d1
<     void X() {
<     }
6a5,7
>     void Y() {
>         int n = 1;
>     }
```

この差分情報を，行番号に基づき解釈すると，「クラス A のメソッド X が削除され，クラス B にメソッド Y が追加されている」となる．このように，従来手法では，実際に行われた細かなプログラム変更に関する情報が欠落してしまう．このような状況において，メソッドの移動や名前変更を検出するためには，第 n 版のクラス A のメソッド X と第 $(n+1)$ 版のクラス B のメソッド Y において，それらの内容の類似度などを計測し (コードクローンと見なせるかどうかを検査して)，その結果から変更を推測するしかない．このように，2 つの版間の単一の更新に複数のプログラム変更が混ざり合うと，従来の手法では正確な変更が検出できないことがある．

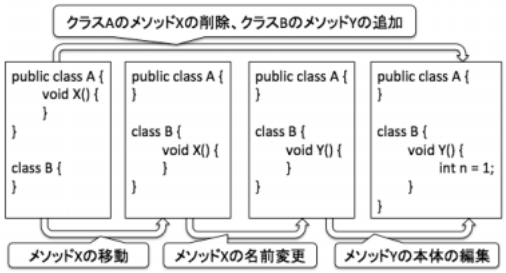


図 1 複数の変更の混在

3 プログラム変更の検出

本章では，編集履歴を用いてプログラム変更を検出する手法を説明する．

3.1 手法の概要

提案手法の概要を図 2 に示す．本手法では，OperationRecorder [1] によって記録された編集履歴が存在することを前提としている．

OperationRecorder で記録される各編集操作には，編集箇所と追加および削除テキストが保存されている．このため，その編集操作を実行する前 (編集前) のソースコードに再適用することで，編集操作を実行した後 (編集後) のソースコードが復元可能である．

本手法では，このような再適用により，プロジェクトに含まれる Java ファイルを復元する．編集操作の適用ごとに，つまり Java ファイルの復元ごとに，プロジェクト全体のスナップショット (編集操作の再

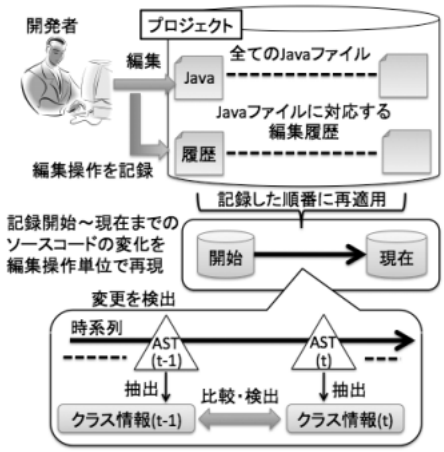


図 2　提案手法概要

適用により復元された Java ファイルと編集操作に影響を受けていない Java ファイルで構成) を保存する．ここでは，時刻 $t1$ に実行された編集操作の再適用により復元されたソースコードを時刻 $t1$ のソースコードと呼ぶ．

次に，それぞれのスナップショットに対して，構文解析を実行し，その成否を検査する．いま，時刻 $t1$ のソースコードが構文解析可能であった場合，そのソースコードから AST(abstract syntax tree) を構築し，クラスメンバに関する情報 (時刻 $t1$ のクラス情報と呼ぶ) を抽出する．構文解析の成否の検査を順次実行していき，時刻 $t2$ のソースコードが構文解析可能であった場合，時刻 $t1$ と $t2$ のクラス情報を比較することで，プログラム変更を検出する．ここで，時刻 $t1$ と $t2$ の間において構文解析可能なソースコードが復元できない場合，時刻 $t1$ と $t2$ のソースコード (あるいはクラス情報) は，構文解析可能かつ連続であると定義する．

3.2　クラス情報

プログラムの変更を検出するために用いるクラスおよびそのメンバに関する情報は，次の 3 種類である．

(1) クラス名　クラスの識別子である．クラスが属するパッケージ名を含む完全限定名 (fully qualified name) で表現する．

(2) フィールド　クラスに含まれるすべてのフィールドの集合を指す．各フィールドは，名前と型に関する情報を持つ．

(3) メソッド　クラスに含まれるメソッドの集合を指す．各メソッドは，名前，戻り値の型，引数の型のリスト，本体コードに関する情報を持つ．

表 1　プログラム変更の分類

種類	内容
AF	フィールドの追加 (Add Field)
DF	フィールドの削除 (Delete Field)
MF	フィールドの移動 (Move Field)
CFN	フィールド名の変更 (Change Field Name)
CFT	フィールドの型の変更 (Change Field Type)
AM	メソッドの追加 (Add Method)
DM	メソッドの削除 (Delete Method)
MM	メソッドの移動 (Move Method)
CMN	メソッド名の変更 (Change Method Name)
CMT	メソッドの戻り値の型の変更 (Change Method Type)
CMP	メソッドの引数の変更 (Change Method Parameter)
CMB	メソッドの本体の変更 (Change Method Body)

3.3　プログラム変更

3.2 節で述べたクラス情報に基づき，本手法が検出するプログラム変更の分類を表 1 に示す．

それぞれのプログラム変更には，それらを検出した直前に適用された編集操作の時刻が割り当てられる．本手法では，比較するクラス情報の対応をクラス名で判断する．よって，匿名クラスに関するプログラム変更は扱わない．また，クラス名の対応が追跡できないプログラム変更，具体的には，クラスの追加，クラスの削除，クラス名の変更は検出しない．

表 1 に示すそれぞれのプログラム変更を検出する手法を 3.3.1 と 3.3.2 に示す．

3.3.1　クラスメンバの追加/削除/書き換え

クラスメンバの追加 (**AF**, **AM**)，削除 (**DF**, **DM**)，書き換え (**CFN**，**CFT**，**CMN**，**CMT**，**CMP**，**CMB**) に関するプログラム変更は，構文解析可能かつ連続な 2 つのソースコードから抽出した時刻 $t1$ のクラス情報と時刻 $t2$ のクラス情報を比較することで検出する．ここでは，それぞれのクラス情報を CI_{t1} および CI_{t2} と表現する．クラス情報の比較手順は，次の 3 つのステップからなる．

(1)　CI_{t1} と CI_{t2} $(t1 < t2)$ において，その内容が

変更されていない (内容が同一となる) クラスメンバの組を検出する．比較対象がフィールドの場合，その名前と型を比較項目とし，それらの一致を検査する．比較対象がメソッドの場合，その名前，戻り値の型，引数の型のリスト，本体コードを比較項目とし，それらの一致を検査する．ここで検出された組に含まれるクラスメンバは，プログラム変更には関係しない．

(2) CI_{t1} と CI_{t2} において，比較項目が 1 つだけ異なるクラスメンバの組を検出する．検出されたクラスメンバの組において，名前だけが異なるフィールドは **CFN**，型だけが異なるフィールドは **CFT** となる．また，名前だけが異なるメソッドは **CMN**，戻り値の型だけが異なるメソッドは **CMT**，引数の型のリストだけが異なるメソッドは **CMP**，本体のコードだけが異なるメソッドは **CMB** となる．

(3) ステップ (1)(2) で検出対象とならなかったクラスメンバは追加あるいは削除に関係する．CI_{t1} に残ったフィールドは **DF**，メソッドは **DM** となる．また，CI_{t2} に残ったフィールドは **AF**，メソッドは **AM** となる．

3.3.2 クラスメンバの移動

本手法では，あるクラスのメンバが Cut & Paste 操作により別のクラスへ移された場合を，クラスメンバの移動 (**MF**, **MM**) として検出する．

(1) 編集履歴から Cut 操作と Paste 操作を抜き出し，それぞれの編集時刻 ($t1$ と $t2$ とする) に基づき，連続する Cut 操作と Paste 操作の組を検出する．連続するとは，$t1 < t < t2$ となる時刻 t において Cut/Copy/Paste 操作が存在しないことを指す．

(2) ステップ (1) で検出した Cut 操作と Paste 操作の組に対して，Cut 操作の直前の構文解析可能なソースコード (直前のクラス情報) において削除されたクラスメンバと，Paste 操作の直後の構文解析可能なソースコード (直後のクラス情報) において追加されたクラスメンバを見つける．

(3) ステップ (2) で見つけた削除および追加された 2 つのクラスメンバに対するコード片 (ソースコード中でのテキスト文字列) が一致し，そのテキスト文字列が Cut 操作において削除される文字列および Paste 操作において追加される文字列にそれぞれ含まれる場合，削除および追加された 2 つのクラスメンバを同一とみなす．このようなクラスメンバは，3.3.1 の手順において，クラスメンバの削除および追加として検出されているので，それら 2 つの変更を取り消し，クラスメンバの移動として合成する．

4 実験

計算機科学を専攻する学生 5 名 (4 名の大学院生と 1 名の学部生) が同じ題材のプログラムを作成した際に記録した編集履歴を用いて，プログラム変更を検出する簡単な実験を行った．作成したプログラムは小規模な GUI ゲームアプリケーションである．5 名の学生 (それぞれ A, B, C, D, E とおく) が作成したプログラムのソースファイルの数，ソースファイルの行数，記録された編集操作の数を表 2 に示す．

4.1 プログラム変更検出ツール

本実験のために，Eclipse 上で動作する，プログラム変更検出ツールを実装した．このツールはプログラムの編集履歴を入力とし，検出したプログラム変更の一覧をファイルへ出力する．クラス情報の抽出における AST の構築には，Eclipse の提供する JavaParser を利用している．

まず，このツールが 2 章に示したように，メソッドの移動と名前変更を分離してプログラム変更を検出可能なことを確認した．実際には，図 1 に示す変更前 (第 1 番目の変更前) ソースコードを 2 つのファイル (A.java と B.java) として用意し，2 章で述べた 3 つの変更を実行した際の編集操作を記録した．メソッドの移動は Cut & Paste 操作で実現した．この編集履

表 2 作成したプログラムに関する情報

学生	A	B	C	D	E
ソースファイル数	27	16	24	25	35
コード行数	2,360	1,267	2,959	1,371	1,483
編集操作数	15,925	7,307	24,478	8,943	16,211

表 3

変更の種類	変更の内容
MM	(cut: 2011/07/27 15:58:10,A) ⇒ (paste: 2011/07/27 15:58:16,B) [method: A.X():void]
CMN	(time: 2011/07/27 15:58:18) (B.X():void ⇒ B.Y():void)
CMB	(time: 2011/07/27 15:58:23) [B.Y():void]

歴からプログラム変更を検出した際の出力を表 3 に示す.

この結果を見ると，3 つのプログラム変更が正確に検出できていることが分かる.

4.2 実験結果

表 2 の編集履歴を用いることで検出したプログラム変更の数を表 4 に示す．A, B, C, D, E は，表 2 におけるそれぞれの学生を指す．AST の数は，各学生のソースコード編集期間において，構文解析可能であったソースコードのスナップショットの数を指す．検出時間は編集履歴の読み込みから検出結果の書き出しまでにかかった秒数を指す．実験には，Intel Core 2 Duo 3.16 GHz の CPU と 2GB の RAM を搭載した計算機を用いた.

表 4 を見ると，編集操作がもっとも多い学生 C の場合，約 18 分検出時間がかかっている．そこで，実験に用いた検出ツールの実行時間を細かく調査した．その結果，クラスメンバの移動先を区別するために作成した，プロジェクト内の継承関係の解析に時間がかかっていることがわかった．ここで，表 4 に示す変更の種類だけを知りたい場合にクラスメンバの移動先情報は不要である．このため，表 4 では継承関係の解析機能を含む場合と含まない場合の検出時間をそれぞれ示している．現実のソースコードに対する検出時間の妥当性や検出時間の短縮については，今後の検討課題である.

次に，検出されたプログラム変更の数がもっとも多い学生 C の編集履歴を用い，編集操作間隔でのプログラム変更の検出と擬似コミット間隔でのプログラム変更の検出との違いを考察する．編集操作間隔でのプログラム変更の検出とは，本論文で提案しているように 1 回の編集操作ごとに構文解析の可否を検査し，

表 4　プログラム変更の検出結果

変更の種類	A	B	C	D	E
AF	192	52	193	63	184
DF	41	5	67	10	82
MF	7	0	8	1	12
CFN	180	29	148	23	152
CFT	23	6	27	14	39
AM	225	127	218	148	284
DM	67	26	69	35	130
MM	7	0	1	9	8
CMN	115	3	44	63	88
CMT	79	11	26	37	60
CMP	104	11	59	39	63
CMB	3,261	902	6,163	1,882	3,711
変更数の合計	4,301	1,172	7,023	2,324	4,813
AST の数	7,364	1,340	9,674	3,316	6,589
検出時間 (継承あり)	41	12	1,057	50	455
検出時間 (継承なし)	40	11	56	23	29

より細かい時間間隔で変更を検出することを指す．これに対して，擬似コミット間隔でのプログラム変更検出とは，版管理システムに開発者がコミットするタイミングを擬似的に設定したものである．ここでは，構文解析可能なソースコードのファイルがセーブ (保存) された場合を，版管理システムへのコミットとみなした．一般的には，構文エラーを含むソースコードや未保存のソースコードをコミットすることはないので，実際のコミットは擬似コミットに包含されるとみなしてよい．つまり，実際の開発におけるコミット間隔は，擬似コミット間隔より長くなるといえる.

学生 C の編集履歴に対するプログラム変更の検出結果を表 5 に示す.「編集操作間隔」の欄の数値は，表 4 の数値と同じである．また,「擬似コミット間隔」の欄における括弧内の数値は，編集操作間隔および擬似コミット間隔で検出した際の変更数の合計が等しくなるように,「擬似コミット間隔」の数値を正規化したものである.

表 5 の変更数の合計を見ると，編集操作間隔での変更数が擬似コミット間隔での変更数の約 2.2 倍 $(7,023/3,133)$ になっている．このことより，編集操作間隔で検出を行うことで，より多くの変更を検出していることが分かる．また，編集操作間隔で検出した際の変更数と擬似コミット間隔で検出した際の正規化後の変更数に着目する．クラスメンバの追加 (**AF**,

表 5 学生 C に関するプログラム変更の検出結果

変更の種類	編集操作間隔	擬似コミット間隔	(正規化)
AF	193	154	(345)
DF	67	72	(161)
MF	8	0	(0)
CFN	148	19	(43)
CFT	27	19	(43)
AM	218	220	(493)
DM	69	102	(229)
MM	1	0	(0)
CMN	44	12	(27)
CMT	26	7	(16)
CMP	59	25	(56)
CMB	6,163	2,503	(5,611)
変更数の合計	7,023	3,133	(7,023)
AST の数	9,674	3,801	—

AM) や削除 (**DF**, **DM**) に関して，それぞれの数値を比較してみると，これら 4 つのすべての変更において編集操作間隔の方の数値が小さくなっている．このような結果の理由として，擬似コミット間隔における変更の検出では，クラスメンバの書き換えや移動が正確に検出できず，クラスメンバの新規追加と削除と検出されてしまったためであると推測できる．

5 おわりに

本論文では，プログラムの変更の把握を支援するために，編集履歴を用いてプログラム変更を検出する手法を提案した．編集操作を再適用することで得られるソースコードの構文情報を用いることで，開発時の変更により近い変更を検出することができた．

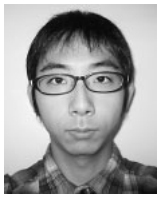
また，簡単な実験を行い，従来よりも検出される変更の数が多くなること，さらに，クラスメンバの追加や削除がクラスメンバの移動や名前変更として検出できている可能性を述べた．

今回の実験を通して，メソッド本体 (**CMB**) に関する変更が非常に多いことが明確になった．プログラム変更の理解を促進するためには，**CMB** をさらに分類していくことが必要である．ただし，検出された変更数が多すぎると逆に理解を妨げる可能性がある．今後，検出する変更の粒度と変更数を調査するための実験をさらに行い，プログラム変更の理解に役立つ情報を明らかにしていくことが重要である．また，本論文では，擬似コミット間隔で実験を行ったが，本手法の有用性を示すためには，実際の開発におけるコミット間隔での評価実験が必須である．

謝辞 本研究の一部は，文部科学省科学研究費補助金基盤研究 (C)(研究課題番号：21500043) による．

参 考 文 献

[1] 大森隆行, 丸山勝久：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，Vol. 49, No. 7 (2008), pp. 2349–2359.
[2] Ren, X., Shah, F., Tip, F. and Ryder, B. G. and Chesley, O. ：Chianti: A Tool for Change Impact Analysis of Java Programs, in *Proc. OOPSLA*, 2004, pp. 432–448.
[3] Robbes, R. and Lanza, M. ：Spyware: A Change-aware Development Toolset, in *Proc. ICSE*, 2008, pp. 847–850.
[4] Robbes, R. and Lanza, M. ：A Change-based Approach to Software Evolution, ENTCS, Vol. 166 (2007), pp. 93–109.
[5] Ryder, B. G. and Tip, F. ：Change Impact Analysis for Object-Oriented Programs, in *Proc. Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 46–53.



木津栄二郎
2010 年立命館大学情報理工学部情報システム学科卒．2010 年より同大学大学院理工学研究科情報理工学専攻修士課程に所属．

大 森 隆 行
2008 年立命館大学大学院理工学研究科博士課程後期課程修了．同年立命館大学助手．2010 年同大学助教，現在に至る．

丸 山 勝 久
1993 年早稲田大学大学院理工学研究科修士課程修了．同年日本電信電話株式会社入社．2000 年 4 月より立命館大学理工学部助教授．2007 年 4 月より同大学情報理工学部教授．博士 (情報科学)．