プロジェクト全体の抽象構文木構築によるファイル間の移動コード検出

藤本　章良[†]　　肥後　芳樹[†]　　松本淳之介[†]　　楠本　真二[†]

Detecting Source Code Moves across Files by Using a Project-Level Abstract Syntax Tree

Akira FUJIMOTO[†], Yoshiki HIGO[†], Junnosuke MATSUMOTO[†], and Shinji KUSUMOTO[†]

あらまし　ソフトウェア開発において，開発者がソースコードの差分を理解することは重要である．差分を検出するツールとして GumTree がある．GumTree は変更前後のソースファイルを入力として受け取ると抽象構文木を生成し，削除・挿入・移動・更新といった操作を抽象構文木のノード単位で検出する．しかし，GumTree は単一ファイルの差分しか検出できないため，ファイルを横断するソースコードの移動を検出できない．本研究では，GumTree を拡張しファイルを横断するソースコードの移動を検出する手法を提案する．提案手法では，プロジェクトに含まれる全てのソースファイルから一つの大きな抽象構文木を構築し，検出精度向上のために抽象構文木に対して 2 段階のマッチングを行う．8 個のオープンソースソフトウェアに対して提案手法を用いて実験を行った結果，88,848 個のコミットの中から合計で 89,418 個のファイルを横断する移動を検出でき，ファイルを横断するソースコードの移動やファイル名に幾つかの特徴が得られた．また既存ツールと比較を行った結果，既存ツールを上回る数の移動を検出した．

キーワード　差分，GumTree，抽象構文木

1. まえがき

ソフトウェア開発において，バージョン管理ツールは多くのプロジェクトで導入されており [1]，バージョン間のソースコードの差分を正確に，理解しやすく開発者に示すことは重要である．開発者がソースコードの差分を理解することで，振る舞いの変化を把握しやすくなるからである [2], [3]．例えばある変更の後にバグが発生した場合，差分を確認することでそのバグの原因となるソースコードの特定が容易となる．そのためソースコードの差分を検出するツールが開発されており，Myers のアルゴリズム [4] を実装した Unix の diff をはじめ，様々な差分検出ツール [5]〜[7] が開発されている．バージョン管理ツールの一つである Git では，内部に diff コマンドが組み込まれている．

しかし，Unix の diff に代表される行単位の差分検出ツールには二つの課題が存在する．一つ目の課題は，出力される差分の粒度が粗いという点である．diff は行単位でソースコードを比較するため，ある行の一部分のみが変更された場合でも，その行全体が変更されたと出力する．このため，実際にどの部分に対して変更が行われたかが分かりづらくなる [5], [8]．diff の出力結果に基づいて，より詳細に変更箇所を強調表示するツール(注1)も存在するが，同じ行の中で複数箇所の変更が発生すると変更箇所以外も強調表示される可能性がある．二つ目の課題は，編集操作が削除と挿入の 2 種類しか存在していない点である．全ての編集操作が削除または挿入の操作として出力されるため，ソースコードの移動や更新といった削除と挿入以外の操作を行った場合，開発者が意図した操作として出力されない．

これらの課題を解決するためのツールとして，抽象構文木（以下，AST）を利用した差分検出ツール [3], [9]〜[11] が開発されている．その一つとして，GumTree [10] がある．GumTree は，ある変更前後のソースファイルを入力として受け取ると，それぞれの AST を生成し，それらを比較して AST のノード単位の差分を出力す

[†] 大阪大学，吹田市
Osaka University, Suita-shi, 565–0871 Japan
DOI:10.14923/transinfj.2020PDP0027

（注1）：diff-highlight．https://github.com/git/git/tree/master/contrib/diff-highlight

るツールである．AST を利用することで行単位での比較よりも細かい粒度で比較可能となり，変更が行われた範囲のみの差分を出力可能である．更にノードの削除や挿入以外にも，移動や更新を検出できる．

GumTree 及び，GumTree を改良したツールが出力した編集スクリプトは多くの研究で利用されている．例えば，Maven のビルドファイル解析 [12] やバグ修正パターン検出の自動化 [13]，バージョン管理のコミットメッセージ生成 [14]，API メソッドの提案 [15] などに用いられている．

しかし，GumTree にも課題が存在する．各ファイルに対して個別に差分計算を行うため，ファイルを横断するソースコードの移動を検出できない．ファイルを横断するソースコードの移動は，リファクタリングにおいて頻繁に行われる [16]．「移動」と出力されるべきこのような変更が，「削除と挿入」として出力されてしまい，その結果，開発者がソースコードの変更に対して誤った認識をもつ可能性がある．

そこで，ファイルを横断するソースコードの移動を検出可能にすることを本研究の目的とする．そのために GumTree を拡張し，複数のソースファイルから生成した AST を一つのプロジェクト全体の AST にまとめ，差分を計算する手法を提案する．提案手法では，計算時間短縮のために必要なソースファイルのみから AST を生成し，検出精度向上のためにプロジェクト全体の AST に対し 2 段階のマッチングを行う．

提案手法の評価を行うために，8 個のオープンソースソフトウェアに対して実験を行い，88,848 個のコミットの中から合計で 89,418 個のソースファイルを横断する移動を検出できた．加えて，それらが差分の理解につながる移動であるかどうかの調査を行った．また，既存のリファクタリング検出ツールに対し，ファイルを横断するソースコードの移動の数を比較した．比較対象としたツールは，リファクタリング操作の一つとしてソースコードの移動を検出可能であり，提案手法との検出結果の違いを調査した．更に，提案手法が検出したファイルを横断するソースコードの移動を目視で確認し，ソースコードがファイルを横断して移動する際に，ソースコードやファイル名にどのような特徴があるかを分析した．

2. 準　　備

2.1 抽象構文木

抽象構文木(注2)は，ソースコードを構文解析して得られる順序木である．一つのソースファイルから一つの AST が生成される．AST の各ノードは以下の五つの要素で構成されている．

ID　各 AST 内で固有の識別子

親ノード　AST の各ノードは，親ノードへの参照をもつ．ただし，根ノードの親は存在しないので何も保持しない．

子ノード　AST の各ノードは，子ノードへの参照をもつ．ただし，葉ノードの子は存在しないので何も保持しない．

ラベル　if 文や変数宣言といった文法上の型を表す．

値　各ノードがもつラベル以外の情報である．例えば，識別子のノードはメソッド名や変数名を値として保持する．

AST のエッジは，その両端のノードに直接の親子関係があることを示す．

2.2 GumTree の差分検出

GumTree は入力された変更前後のソースファイルからそれぞれの AST を生成し，二つの AST の違いを AST のノード単位の編集スクリプトとして出力する．編集スクリプトとは，変更後の AST を得るために変更前の AST に適用された編集操作の列である．木構造の差分を計算するために，GumTree はマッチングを行う．マッチングとは，変更前後における AST のノード間に対応付けを行う処理である．

GumTree のマッチングは，トップダウンフェーズとボトムアップフェーズの 2 段階で構成されている．トップダウンフェーズでは，二つの AST の根ノードから葉ノードに向けて辿っていき，完全一致する部分木をマッチングする．ボトムアップフェーズでは，トップダウンフェーズでマッチングされていないノードに対して類似度を用いてマッチングを行う．ボトムアップフェーズは葉から根に向かって辿り，根ノードにたどり着くと完了となる．

マッチングされたノードは，変更の前後で同じノードとして扱われる．マッチングの結果と AST を参照し，変更前の AST に対して削除・挿入・移動・更新が行われたノードを得る．ただし，AST の類似度をもと

(注2)：Abstract Syntax Tree: AST

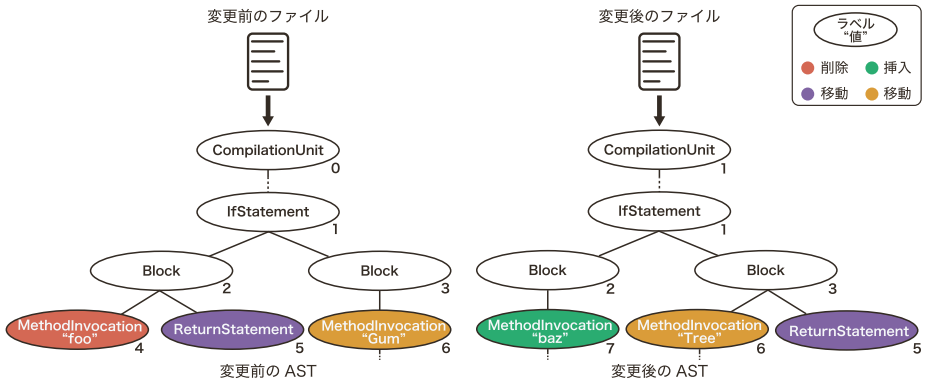


図 1　GumTree の差分検出

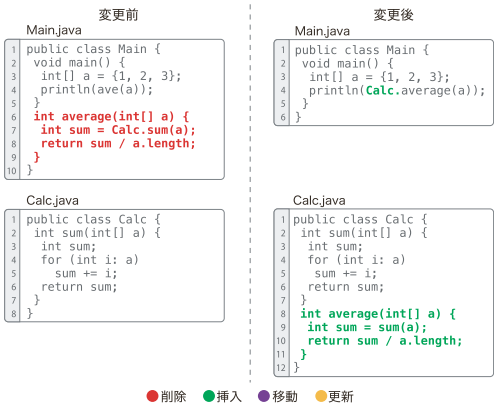
にマッチングを行うため，必ずしも開発者が行った操作と一致するとは限らない．

図 1 は，GumTree の差分検出方法を模式的に表している．変更前後のファイルから生成した AST に対してマッチングを行う．図 1 では，マッチングの結果を各ノードの右下の数字で表しており，同じ数字のノードはマッチングされていることを示している．変更前後の AST ノードのマッチング結果を基に Chawathe ら [17] の計算手法を用いて，GumTree は木構造の違いを差分として出力する．この例では，4 番のノードは変更前の AST にのみ存在するため削除，7 番のノードは変更後の AST にのみ存在するため挿入と出力される．5 番のノードは変更前後のどちらにも存在しているが，親のノードが変わったため移動と出力される．6 番のノードは木構造上の違いはないが，ノードの値が “Gum” から “Tree” に変更されたため更新と出力される．

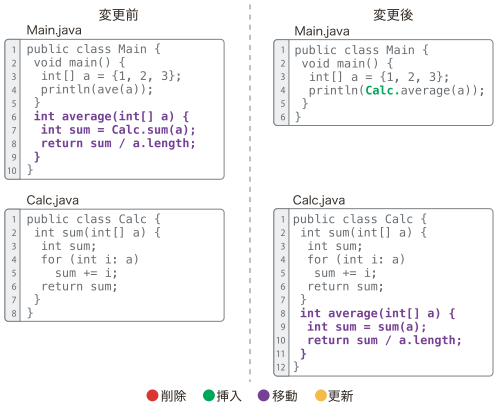
3. 研 究 目 的

GumTree は，変更を削除・挿入・移動・更新の四つの操作を用いて表現する．しかし，AST は各ファイルに対して構築されるため，ソースファイル A からソースファイル B にコードが移動される変更が行われた場合，その変更を移動であると認識できない．そのような場合に GumTree は，ソースファイル A からはコードが削除され，ソースファイル B にはコードが追加されたという編集スクリプトを出力する．

図 2 の例では，ファイル `Main.java` と `Calc.java` が同時に変更されたと仮定する．この変更では，ファイル `Main.java` に存在したメソッド `average()` が `Calc.java` に移動されている．この変更の前後で，



(a) GumTree の出力



(b) 実際に行われた操作をより適切に反映する出力

図 2　実際の編集操作と GumTree の出力が異なる例

GumTree を用いてそれぞれのファイルの差分を出力すると，`Main.java` の編集スクリプトはメソッド `average()` の削除，`Calc.java` の編集スクリプトは

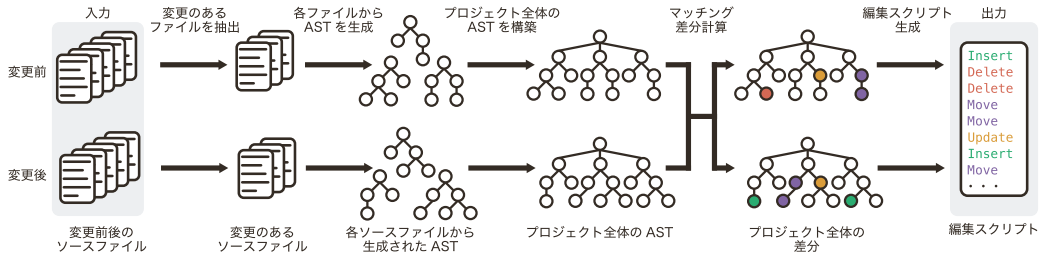


図 3　提案手法の概要

メソッド average() の挿入となる（図 2 (a)）．しかし，実際に行われた操作を表現するには，メソッド average() の移動とする方が適切である（図 2 (b)）．この違いにより，開発者がソースコードの変更について誤った認識をもち，ソースコードの振る舞いの違いを理解するのに時間がかかったり，バグの発見が遅れたりするおそれがある．本研究では複数のファイルが変更された場合に，ファイルを横断する移動を検出する手法を提案する．

4. 提 案 手 法

提案手法の概要を図 3 に示す．提案手法の入力は，対象のプロジェクトに含まれる変更前後のソースファイル全てであり，出力は編集スクリプトである．まず，入力として与えられた各ソースファイルから AST を生成する．それらの AST を用いて，プロジェクト全体の AST を構築する．このプロジェクト全体の AST は，変更前と変更後で一つずつ作られる．プロジェクト全体の AST の構築方法は，**4.1** で述べる．

次に，構築したプロジェクト全体の AST に対してマッチングを行う．その結果を用いて，プロジェクト全体の差分を計算し出力する．プロジェクト全体の AST の差分計算には GumTree の差分を計算する処理をそのまま再利用し，編集スクリプトを出力する．

しかし，単純にプロジェクト内の全てのソースファイルから単一の巨大な AST を構築し差分を計算すると，計算に多大な時間を要したり差分を検出する精度が下がったりする．そこで，**4.2** 及び **4.3** の工夫により，これらの問題を回避する．

4.1 プロジェクト全体の AST の構築

プロジェクト全体の AST の構築を図 4 に示す．まず，プロジェクト全体の AST を構築するため，ノードを一つ作成する．このノードを根ノードとし，ソースファイルから生成した AST を子ノードとして加えていく．ただし AST を加える順番は，プロジェクトルートディレクトリからそのファイルへのパス名の昇順とする．

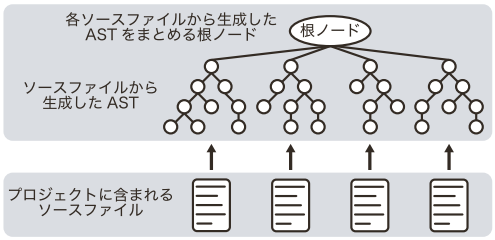


図 4　プロジェクト全体の AST の構築

4.2 変更のないファイルの除外

プロジェクトに含まれる全てのソースファイルからプロジェクト全体の AST を構築すると，差分の計算対象となる AST ノードの数が増え，提案手法の実行時間が長くなってしまう．また，変更されていないファイルも検出対象とすることで，無変更であるにもかかわらず操作が行われたと誤って検出する可能性がある．そこでプロジェクト全体の AST を構築する段階で，ソースファイルごとに変更の有無を確認する．変更されたソースファイルのみから AST を生成し，プロジェクト全体の AST に加える．

ファイル内容の変更の有無は Myers のアルゴリズム [4] を用いて調べる．得られた結果が以下のいずれかに該当する場合は，そのファイルからは AST を生成しない．

- 全く変更がない
- 空行・スペース・タブの削除，挿入のみ

4.3 2 段階のマッチング

差分の検出対象をファイルからプロジェクト全体に拡大した場合，変更されていないにもかかわらず類似したソースコード間で移動と出力される場合がある．この原因は，AST がプロジェクト全体に拡大し，

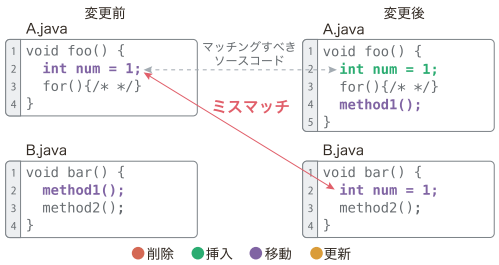


図 5　不適切なマッチングによる誤検出の例

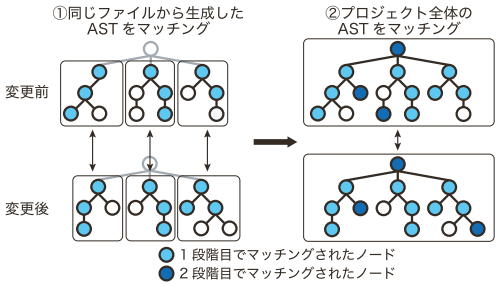


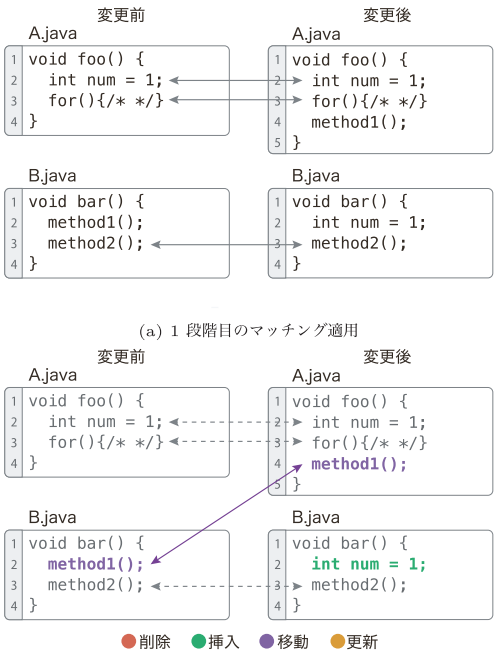
図 6　2 段階のマッチング

GumTree がノードを適切にマッチングできていないためである．誤検出の例を図 5 に示す．この例では，変更の前後で `A.java` の 2 行目に書かれた `int num = 1;` は変更されていない．しかし図中の赤線で示しているように，`A.java` の `int num = 1;` が `B.java` の `int num = 1;` に不適切にマッチングされた．その結果，`A.java` の `foo()` から `B.java` の `bar()` に移動したと誤検出している．

適切にマッチングできない理由は，マッチング対象がプロジェクト全体に拡大されてマッチングするノードの候補が増えるためである．そこで適切にマッチングするために，2 段階のマッチングを行う．2 段階のマッチングの方法を図 6 に示す．まず，プロジェクト全体の AST の中で同じファイルから生成した部分木を取り出して，マッチングを行う．その後，プロジェクト全体の AST に対してマッチングを行う．

1 段階目では，変更前後で同一ファイルから生成した AST ノードのマッチングを行う．これは AST ノードのマッチング候補を同一ファイルのみにすることで，マッチング候補の数を減らし，適切にマッチングを行うことが狙いである．また，この段階でのマッチングは単一ファイルの変更前後の AST をマッチングしており，既存手法にあたる．

2 段階目のプロジェクト全体のマッチングでは，1 段階目のマッチング終了時にまだマッチングされていないノードの中からマッチングを行う．これにより既存手法ではマッチングできなかった，異なるファイル間の AST ノードをマッチングできる．また，マッチングするノードの候補を 1 段階目でマッチングされていないノードに限定することにより，類似したソースコードが移動と誤検出される可能性を下げられる．



(a) 1 段階目のマッチング適用

(b) 2 段階目のマッチング適用

図 7　2 段階のマッチングを適用した際の出力

図 5 の変更に対して，2 段階のマッチングを適用すると，1 段階目の同一ファイルのマッチング時に，変更前後の `A.java` に含まれる `int num = 1;` がマッチングされる（図 7 (a)）．この結果，`A.java` の `int num = 1;` に対する変更は行われていないと出力される．また，2 段階目のマッチングでは変更前の `B.java` に含まれる `method2()` と変更後の `A.java` に含まれる `method2()` がマッチングされる（図 7 (b)）．これにより，ファイルを横断する移動を検出できる．

5. 実　　験

提案手法の評価や提案手法が出力した結果を分析するため，提案手法を実装した Graftast (注3) を用いて複数のオープンソースソフトウェア（以下，OSS という）

(注3)：https://github.com/kusumotolab/Graftast

表 1　実験対象の OSS プロジェクト

OSS 名	コミット数	実験対象範囲の最終コミット日
ArgoUML	16,144	2015 年 1 月 11 日
dnsjava	1,771	2019 年 10 月 27 日
Eclipse (注4)	29,811	2019 年 11 月 17 日
JHotDraw	763	2018 年 8 月 27 日
JUnit 4	2,418	2019 年 11 月 2 日
Apache Log4j 2	10,752	2019 年 11 月 1 日
Apache Struts	5,697	2019 年 11 月 4 日
Apache Tomcat	21,492	2019 年 11 月 6 日

表 2　検出したファイルを横断する移動の数

OSS 名	ファイル間の移動の数	全ての移動の数
ArgoUML	10,316 (23.1%)	44,707
dnsjava	1,664 (7.3%)	22,851
Eclipse	28,112 (22.0%)	127,693
JHotDraw	11,800 (27.9%)	42,239
JUnit 4	3,365 (25.0%)	13,442
Apache Log4j 2	8,031 (15.2%)	52,678
Apache Struts	4,042 (14.9%)	27,093
Apache Tomcat	22,088 (19.2%)	114,863

表 3　既存手法と提案手法の増減数

OSS 名	削除	挿入	移動	更新
ArgoUML	−16,852	−16,852	3,471	800
dnsjava	−1,461	−1,461	413	66
Eclipse	−35,006	−35,006	6,450	1,110
JHotDraw	−10,212	−10,212	2,265	375
JUnit 4	−3,713	−3,713	671	121
Apache Log4j 2	−8,967	−8,967	1,650	270
Apache Struts	−6,474	−6,474	1,330	185
Apache Tomcat	−50,693	−50,693	12,223	1,138

に対して実験を行った．以下では，この実験の内容について説明する．

5.1 実験対象

GumTree の評価実験に用いられたデータセット [18] に含まれている OSS のうち，Git に移行されているソフトウェアを実験対象とした．実験対象の OSS を表 1 に示す．これらのプロジェクトは全て Java で開発されている．

5.2 実験 1 ファイルを横断するソースコードの移動検出

提案手法によってファイルを横断するソースコードをどの程度検出できるかを調査する．Git でバージョン管理されている OSS に対し，あるコミットに含まれるソースファイルを変更後のソースファイル，その親コミットに含まれるソースファイルを変更前のソースファイルとして，提案手法へ入力し差分を検出する．これを master ブランチ上に存在する全てのコミットに対して適用する．検出した差分のうち，ファイルを横断する移動の数を計測する．

また，既存手法と提案手法で編集スクリプトがどのように変化したのか調査する．既存手法には，変更前後のプロジェクトにおいて全ての同名ファイルの組を入力として与え，得られた編集スクリプトから各編集操作の数を合計する．

各 OSS ごとに，ファイルを横断するソースコードの移動の検出数，検出された全ての移動数を表 2 に示す．ファイル間の移動数の横に示されている括弧は，検出された全ての移動の数に対するファイルを横断する移動の数の割合である．全てのプロジェクトにおいて，ファイルを横断する移動が含まれている．その割合はプロジェクトによって 7.3%〜27.9% とばらつきがある．

既存手法との比較結果を表 3 に示す．各項目は提案手法の検出数から既存手法の検出数を引いた値である．全てのプロジェクトにおいて，削除・挿入の検出数が減少し，移動の数が増加している．これは，図 2 のように既存手法では削除と挿入となる出力が，提案手法によってファイルを横断する移動として出力された結果である．削除・挿入の減少数が移動の増加数と一致していないのは，複数の削除・挿入されたノードをまとめて一つの移動と検出したためである．

また表 2 の “ファイル間の移動の数” と表 3 の “移動” の数が異なっているが，これは検出対象となるファイルが異なっているためである．既存手法との比較において，条件を揃えるため提案手法の検出対象を制限している．それぞれの検出対象は以下のとおりである．

- 提案手法による移動検出（表 2）：プロジェクト内の全てのソースファイル
- 既存手法との比較（表 3）：変更前後の両方に存在するソースファイル

5.3 実験 2 ファイルを横断するソースコード検出の精度

提案手法によって検出できたファイルを横断する移動に対して，その精度を評価する．提案手法が検出したファイルを横断する移動の中には，図 8 のように移動前後のソースコード片が大きく異なっている場合がある．これは，`for` 文の移動に加えて，その内部に対して `if` 文の挿入やステートメントの挿入・削除といっ

(注4)：org.eclipse.ui.workbench のみ調査した．Eclipse はプロジェクトの規模が非常に大きく，CVS-Vintage では eclipse.ui.workbench と eclipse.jdt.core のみ調査している．我々は eclipse.jdt.core に対しても実験を行ったが，実行にかかる時間が長すぎたため実験対象から除外した．

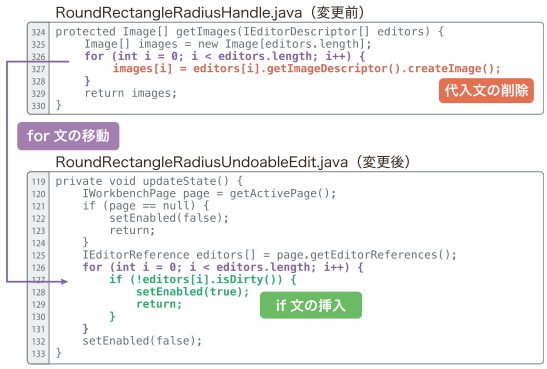


図 8　移動元と移動先のソースコードが大きく異なる例

た複数の操作の組み合わせとして検出された結果である．検出されたそれぞれの操作は誤りではないが，for 文の移動のみに注目した場合，この検出結果が適当であるとはいえない．

そこで，提案手法が検出したファイルを横断する移動に対し移動前後のソースコード片の類似度を用いて精度の評価を行う．例えば図 8 の場合，移動前後のソースコード片が大きく異なっており類似度は低い．一方で類似度が高い場合，二つのソースコード片が適切にマッチングし正しく移動を検出した可能性が大きい．したがって，類似度が高い移動が多いほど提案手法が検出する移動の精度が高いと判断する．

この実験においてはソースコード片の類似度を

$$1 - \frac{\text{GumTree で計算した編集スクリプトの長さ}}{\text{変更前後の移動コードのノード数合計}}$$

と定義した．移動前後のソースコード片が完全に一致した場合，編集スクリプトの長さは 0 となり類似度は 1 になる．反対に大きく異なっていた場合は移動前後のソースコード片の編集スクリプトの長さが長くなるため類似度が低下する．

提案手法が検出した全てのファイルを横断する移動に対し類似度を計算し，その結果から得られたヒストグラムが図 9 である．類似度が 1，すなわち移動前後のコード片が完全に一致した移動が全体の 80% 以上を占めていることがわかる．また約 93% が類似度 0.7 以上となっており，提案手法の移動検出の精度は十分であると考える．

5.4　実験 3 差分の理解につながる事例の調査

実験 1 より，提案手法がファイルを横断するソースコードの移動を多数検出できることが明らかになった．この実験では提案手法が検出したファイルを横断する

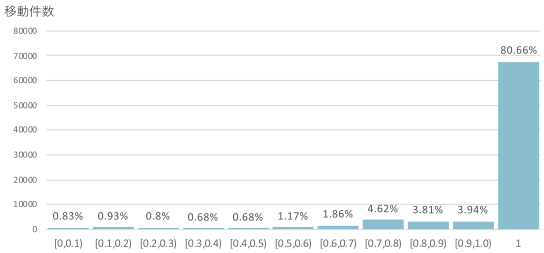


図 9　移動前後のソースコード片の類似度の分布

移動の中で，開発者の差分理解につながる検出結果があるかを既存手法と比較し調査する．

Apache Tomcat で行われたリファクタリングコミットの前後に，WsSession.java と Util.java をそれぞれ既存手法を用いて差分検出を行うと，図 10 の結果が得られた．これは，WsSession.java のメソッドである getMeesageType が削除され，Util.java ではメソッド getMeesageType が挿入されたという出力である．一方，提案手法を用いて差分の検出を行ったところ，図 11 のような結果が得られた．これは，既存手法では削除と挿入として検出されたが，提案手法ではファイルを横断する移動として検出された例である．この結果から開発者は，メソッド getMeesageType は Util.java で新たに作成されたメソッドではなく，WsSession.java から移動されたメソッドであることが理解しやすくなる．

同様に，Apache Tomcat におけるバグ修正コミットの前後で既存手法と提案手法の比較を行った．既存手法を JspServletWrapper.java と JspServlet.java に対して適用したところ，JspServletWrapper.java ではある catch 節の削除，JspServlet.java では，新規コードの挿入とその中の一部が同じファイル内での移動であると出力された（図 12）．同コミットに対し，提案手法を適用すると図 13 の結果が得られた．これは，JspServletWrapper.java で削除されたコードの一部が，JspServlet.java にファイルを横断して移動したと提案手法が新たに検出したことを示している．開発者は，この結果から JspServletWrapper.java で行っていた一部の処理が JspServlet.java で行われるように変更されたことが理解しやすくなる．

これらの例のように，提案手法の検出結果により開発者が差分の理解をしやすくなる事例が複数あり，提案手法は既存手法に比べ有用であるといえる．

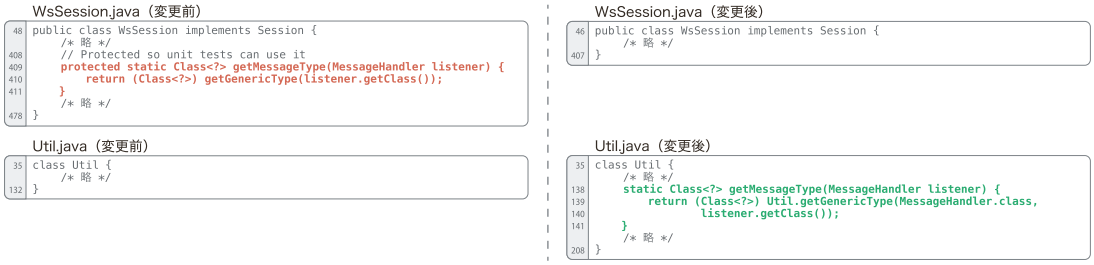


図 10　既存手法によるリファクタリング時の差分検出結果

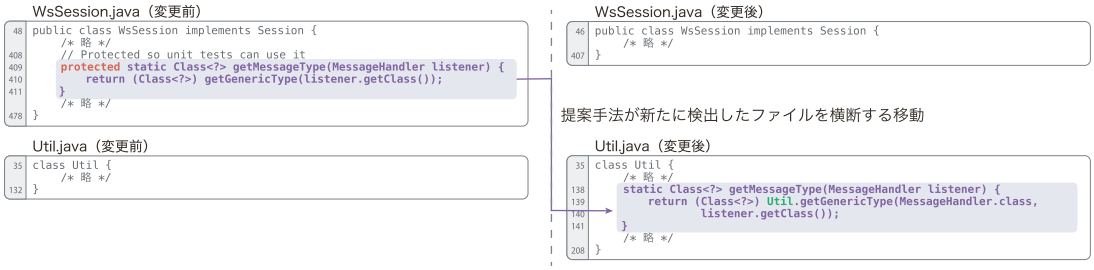


図 11　提案手法によるリファクタリング時の差分検出結果

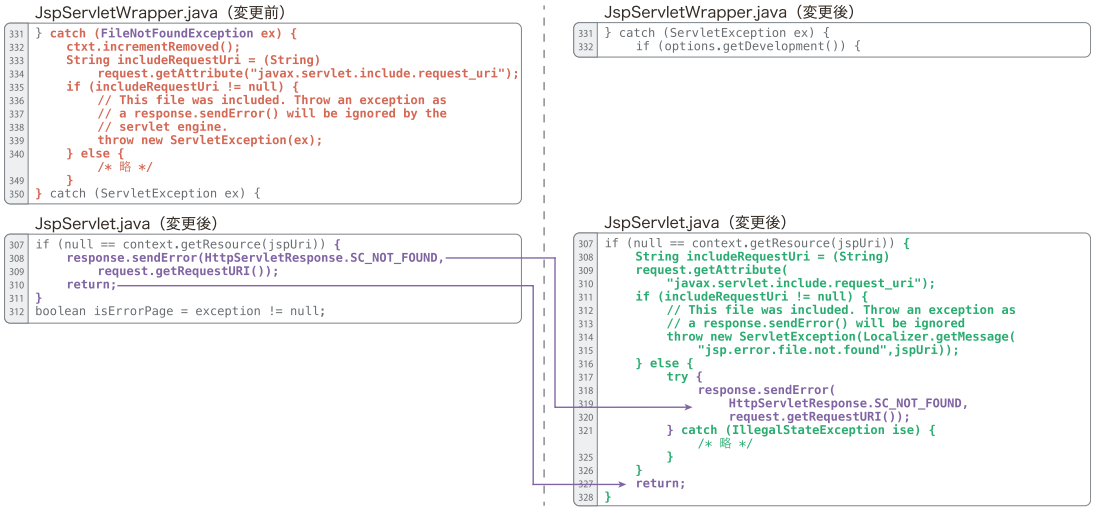


図 12　既存手法によるバグ修正時の差分検出結果

5.5　実験 4 ファイル間の移動コードを検出できる既存ツールとの比較

リファクタリング検出ツール [19]〜[21] の中にはファイルを横断するソースコードの移動やそれに類する操作を検出できるツールがある．同じプロジェクトに対して，提案手法によって得られる検出結果と既存のリファクタリング検出ツールを用いて得られる検出結果を比較し，提案手法がファイルを横断するソースコードの移動検出に対して有効であるかを評価する．

比較対象の既存ツールとしてリファクタリング検出ツールである RefactoringMiner [19] を選択した．このツールは，リファクタリング操作として，メソッド・クラスの移動，メソッドのプルアップ，メソッドへの抽出，パッケージの変更などが検出可能である．

この実験では，RefactoringMiner によって検出された操作のうち，メソッドの移動，メソッドのプルアップ，メソッドのプッシュダウンの数と，提案手法で検出した移動ノードが MethodDeclaration である移動の数を

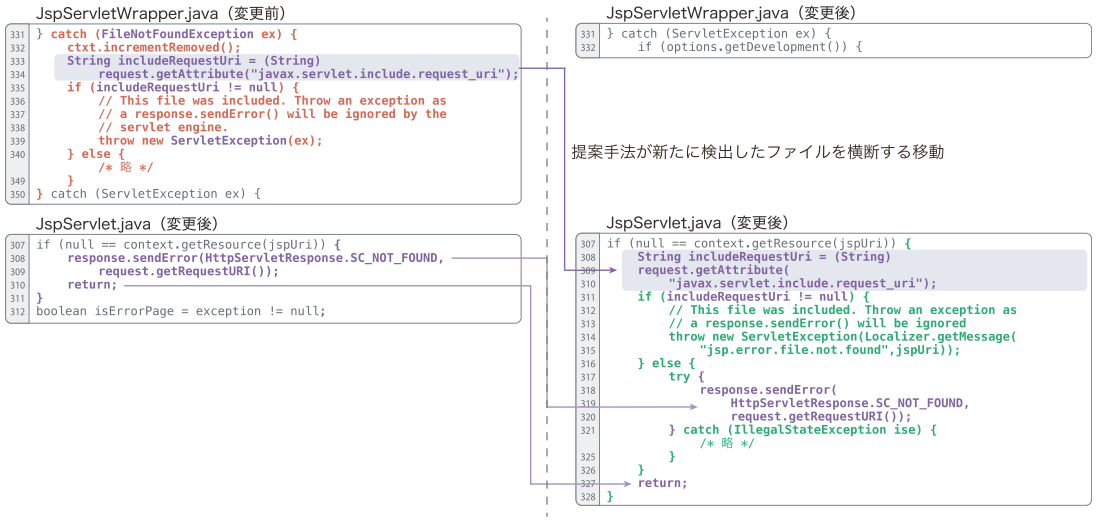


図 13　提案手法によるバグ修正時の差分検出結果

比較する．ただし，RefactoringMiner が検出したメソッドのプルアップやプッシュダウンは複数件のメソッドの移動として出力されるが，この実験ではそれらの移動を 1 件の移動として集計する．これは n 個のメソッドを一つのメソッドに集約した場合，RefactoringMiner では n 件の移動が発生したと出力される一方で，提案手法はメソッドのプルアップやプッシュダウンといった操作を 1 件のメソッドの移動として検出するため，比較の条件を一致させるためである．

この実験は，**5.1** で挙げた OSS のうち eclipse.ui.workbench を除いた OSS を対象にする．RefactoringMiner は，Git リポジトリ全体のリファクタリングを検出するため，一部のモジュールやディレクトリに限定した範囲で行われたリファクタリングの検出ができなかったためである．また，RefactoringMiner の実行時には検出するブランチに master ブランチを指定した．これは，提案手法と検出対象範囲となるコミットを一致させるためである．

各 OSS に対して，提案手法と RefactoringMiner を適用し，検出できたファイルを横断する移動の数を表 4 に示す．表 4 では，提案手法と RefactoringMiner のどちらも検出した移動の数，提案手法のみが検出した移動の数，RefactoringMiner のみが検出した移動の数を記載している．Apache Tomcat 以外のプロジェクトにおいて，提案手法が検出した移動の数が RefactoringMiner の検出した移動の数を上回った．

次に，移動したメソッドが宣言されていたクラスの種類による分類結果を表 5 に示す．匿名クラス内で宣言されたメソッドが通常のクラス内に移動する事例は，提案手法のみで検出された．例として，図 14 のような移動が検出された．また，内部クラスで宣言されたメソッドの通常クラス内への移動は，提案手法や RefactoringMiner どちらも検出できたが，検出数は提案手法の方が多いという結果が得られた．

匿名クラスと通常クラス間でのメソッドの移動が提案手法でのみ検出できたのは，移動の検出方法の違いが理由であると考えられる．RefactoringMiner はルールベースでリファクタリング操作の検出を行う．RefactoringMiner は JDT (注6) の AST を用いて解析を行ってい

表 4　提案手法と RefactoringMiner が検出したファイルを横断するメソッドの移動数

OSS 名	両手法	提案手法のみ	RM (注5) のみ
ArgoUML	148	92	77
dnsjava	53	73	9
JHotDraw	78	223	73
JUnit 4	94	119	39
Apache Log4j 2	124	519	104
Apache Struts	191	117	95
Apache Tomcat	895	315	384

表 5　メソッドが宣言されたクラスによる分類

	両手法	提案手法のみ	RM のみ
匿名クラスのメソッドの移動	0	26	0
内部クラスのメソッドの移動	35	67	37

（注5）：RefactoringMiner の略表記
（注6）：Eclipse プラグイン（https://projects.eclipse.org/projects/ eclipse.jdt）

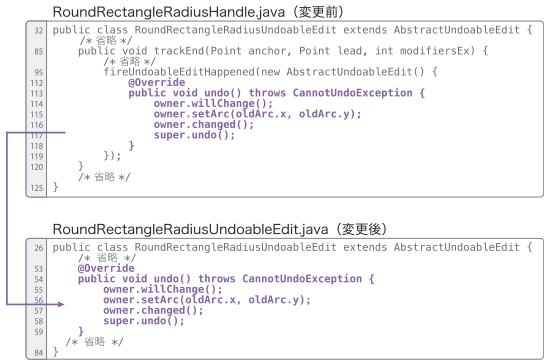


図 14　匿名クラス内のメソッド undo() が通常クラス内に移動する例

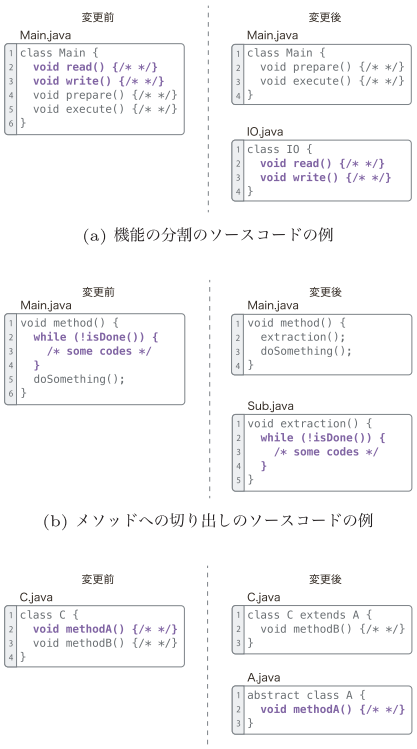
るが，ルールの一つとして，リファクタリング検出対象のメソッドは TypeDeclaration のノード内で宣言されるメソッドのみである．JDT の AST では，匿名クラスは AnonymousTypeDeclaration ノードと対応しており，通常のクラスや内部クラスが対応する TypeDeclaration とは異なる．ゆえに，匿名クラス内のメソッドはリファクタリングの検出対象から外れるため，匿名クラスと通常クラス間での移動は検出できなかったと考えられる．

一方，提案手法で用いる Java ソースファイルから生成される AST も JDT の AST に基づいているが，差分の検出には GumTree のアルゴリズムを用いて移動を検出する．**2.2** で説明したように，このアルゴリズムでノードが移動したと判定されるのは，マッチングされたノードの親が変更前後で異なる場合である．したがって，親のノードの種類が AnonymousTypeDeclaration であっても移動を検出可能である．よって，表 5 のような結果となった．

また，移動したメソッドが通常のメソッドかコンストラクタであるかを調査した．提案手法と RefactoringMiner どちらからも検出された数は 6 個，提案手法のみが検出した数は 137 個，RefactoringMiner のみが検出した数は 12 個であり，コンストラクタの移動はどちらの手法からも検出できるものの，その数は提案手法の方が多いという結果が得られた．

5.6　実験 5 ファイルを横断するソースコードの移動の特徴

提案手法が検出したファイルを横断するソースコードの移動の特徴や傾向を調査する．移動したノードの種類や処理の内容に加え，変更の前後でファイルの内容がどのように変化したかや，移動前後のファイル名の



(a) 機能の分割のソースコードの例

(b) メソッドへの切り出しのソースコードの例

(c) 継承関係の変化のソースコードの例

図 15　ファイルを横断する移動が発生したソースコードの例

特徴，などを調べる．調査するノードは，MethodDeclaration，WhileStatement，ForStatement，IfStatement，Block の五つである．

検出されたファイルを横断する移動の中で特徴的であった移動の例を幾つか紹介する．各図のソースコード例の紫色で示した部分が移動と検出された箇所である．

機能の分割・統合

大きいクラスを複数のクラスに分割するときに，ファイルを横断するメソッドの移動が検出された．その例を図 15 (a) に示す．変更前は `Main` クラスに含まれていたメソッド `read()` と `write()` が `IO` クラスに移動している．このほかにも，似た機能をもつクラスを統合して一つのクラスにするための移動もあった．

移動先のクラスが新たに作られる場合と，既存のクラスにソースコードが移動される場合を確認した．移動前後のファイル名に，Util，Helper といった名称が付く例が合計で 334 個あった．またそれら以外には，元の名前と似た名称になる場合が多かった．

メソッドの移動以外にも，図 15 (b) のような一部の処理のみを取り出してメソッド化する変更があった．WhileStatement，ForStatement，IfStatement のノードでこのような移動が見られた．変更前のメソッドの行数が長く，処理を別のメソッドに切り出すリファクタリングをした場合に，このような移動が多く検出された．

継承関係の変化

クラス間の継承関係が変化する場合に，ファイルを横断する移動を検出した．図 15 (c) は，新たに作成した抽象クラスに一部のソースコードが移動される例である．具象クラスから抽象クラスへのソースコードの移動が多く検出された．反対に，抽象クラスから具象クラスへの移動も存在した．その他にも，継承関係のある具象クラス間で，実装箇所の変更によるソースコードの移動も検出した．

継承関係のあるファイル間での移動は，Abstract という名称がつくファイルの他に，Default という名称がつく例が多く確認された．

検出できたファイルを横断する移動のうち，機能の分割・統合，継承関係の変化，部分的なメソッドへの切り出しといった操作はリファクタリングにおいて多く行われる [22], [23]．典型的なリファクタリングに伴う移動が，提案手法によって検出できた．

またファイルを横断するソースコードの移動の中には，行数が短いソースコードも存在した．次のソースコードが挙げられる．

- getter/setter（MethodDeclaration）
- return 文（Block）
- null チェック（IfStatement）

getter や setter の移動は，抽象クラスを追加した場合によく見られた．

return 文はブロックの中に `return;` のように return 文のみが含まれるソースコードの移動である．このような移動はほとんどのプロジェクトで検出された．

IfStatement で null チェックを行うソースコードの移動も検出された．この if 文内では，新規インスタンスの作成のみや return 文など短いソースコードも含まれている．

このような短いソースコードの移動が多く検出された理由として，これらのソースコードは定型処理としてプログラム内に多く存在していると考えられる．それらをマッチングしたとき，異なるファイル間でのノードのマッチングが多くなったと考えられる．更に，ソースコードの長さも影響がある．GumTree は部分木の類似度を用いて，しきい値が一定値を超えた場合にマッチングを行う．短いソースコードであれば，ソースコードの差異が出にくく類似度が高くなりやすい．そのため，よりマッチングされやすくなるので，移動として検出される可能性が高くなると考えられる．

6. 妥当性の脅威

本研究における妥当性の脅威について述べる．

提案手法は特定のプログラミング言語に依存しないため，GumTree を適用可能なプログラミング言語全てにおいて適用可能である．本研究では，Java で記述されたプロジェクトに対してのみ実験を行った．他の言語でもファイルを横断するソースコードの移動を検出できると予想されるが，実際に検出できるかは分からない．

事例の調査においては，差分の理解につながる移動であるかを目視で確認した．しかし，この結果は著者らの主観に大きく依存する．そのため，差分の理解につながる移動と判断した例が，他の開発者にとっては差分の理解につながらない移動となる可能性がある．

また，提案手法との比較対象として，RefactoringMiner を用いた．その他の既存ツールを用いた場合，検出できるファイルを横断する移動の数や検出された移動の特徴について，異なる実験結果になる可能性がある．

7. むすび

本研究では，各ファイルに対して個別に差分の計算を行う GumTree を拡張し，プロジェクト全体のソースファイルの中からファイルを横断するソースコードの移動を検出する手法を提案した．提案手法では，プロジェクトに含まれる全てのソースファイルから，プロジェクト全体の AST を構築し，その差分を計算する．提案手法を用いて 8 個の OSS に対して実験を行ったところ，合計で 89,418 個のファイルを横断する移動を検出できた．既存ツールと比較を行い，提案手法でのみ検出できる移動のパターンが存在することがわかった．更に移動したソースコードを調査した結果，リファクタリングに伴う移動をはじめ，移動するソースコードの特徴やファイル名の傾向が明らかになった．

今後の課題としては以下が考えられる．

他の言語で開発された OSS への適用

Java 以外の言語で開発されている OSS に対して提案手法を適用し，ファイルを横断するソースコードの

移動が検出できるかを調査する．

検出した差分の可視化

現状では，検出した差分は編集スクリプトとして出力されるが，直感的に差分を理解しづらい．そこで，検出されたファイルを横断するソースコードの移動を，より分かりやすく開発者に表示する．GumTree には，Web ブラウザを用いて編集スクリプトを視覚的に確認できるツールがあり，このツールへの改良が考えられる．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：17H01725）の助成を得て行われた．

文　献

[1] B. De Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?,” ICSE Workshop on Cooperative and Human Aspects on Software Engineering, pp.36–39, 2009.

[2] D. Binkley, “An empirical study of the effect of semantic differences on programmer comprehension,” International Workshop on Program Comprehension, pp.97–106, 2002.

[3] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, “ClDiff: Generating concise linked code differences,” ACM/IEEE International Conference on Automated Software Engineering, pp.679–690, New York, NY, USA, 2018.

[4] E.W. Myers, “An O(ND) difference algorithm and its variations,” Algorithmica, vol.1, no.1, pp.251–266, 1986.

[5] G. Canfora, L. Cerulo, and M. Di Penta, “Ldiff: An enhanced line differencing tool,” IEEE International Conference on Software Engineering, pp.595–598, 2009.

[6] A. Loh and M. Kim, “LSdiff: A program differencing tool to identify systematic structural differences,” ACM/IEEE International Conference on Software Engineering - Volume 2, pp.263–266, 2010.

[7] Z. Xing and E. Stroulia, “UMLDiff: An algorithm for object-oriented design differencing,” IEEE/ACM International Conference on Automated Software Engineering, pp.54–65, 2005.

[8] 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，” 情処学論，vol.49, no.7, pp.2349–2359, 2008.

[9] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” IEEE Trans. Softw. Eng., vol.33, no.11, pp.725–743, 2007.

[10] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” International Conference on Automated Software Engineering, pp.313–324, 2014.

[11] G. Dotzler and M. Philippsen, “Move-optimized source code tree differencing,” IEEE/ACM International Conference on Automated Software Engineering, pp.660–671, 2016.

[12] C. Macho, S. McIntosh, and M. Pinzger, “Extracting build changes with builddiff,” International Conference on Mining Software Repositories, pp.368–378, 2017.

[13] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, “Towards an automated approach for bug fix pattern detection,” 2018.

[14] M.S. Ahmed and A. Tabassum, “Automatic contextual commit message generation: A two-phase conversion approach,” 2018.

[15] A.T. Nguyen, M. Hilton, M. Codoban, H.A. Nguyen, L. Mast, E. Rademacher, T.N. Nguyen, and D. Dig, “API code recommendation using statistical learning from fine-grained changes,” International Symposium on Foundations of Software Engineering, pp.511–522, ACM, 2016.

[16] B. Geppert, A. Mockus, and F. Robler, “Refactoring for changeability: A way to go?,” IEEE International Software Metrics Symposium, 2005.

[17] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” SIGMOD Rec., vol.25, no.2, pp.493–504, 1996.

[18] M. Monperrus and M. Martinez, “CVS-Vintage: A dataset of 14 cvs repositories of java software,” 2012.

[19] N. Tsantalis, M. Mansouri, L.M. Eshkevari, D. Mazinanian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” International Conference on Software Engineering, pp.483–494, 2018.

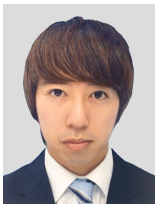
[20] D.C. Atkinson and T. King, “Lightweight detection of program refactorings,” Asia-Pacific Software Engineering Conference, pp.663–670, 2005.

[21] Z. Xing and E. Stroulia, “Refactoring detection based on UMLDiff change-facts queries,” Working Conference on Reverse Engineering, pp.263–274, 2006.

[22] M. Fowler, Refactoring: Improving the design of existing code, Addison-Wesley Professional, 2018.

[23] T. Mens and T. Tourwe, “A survey of software refactoring,” IEEE Trans. Softw. Eng., vol.30, no.2, pp.126–139, 2004.

（2020 年 5 月 29 日受付，9 月 29 日再受付，
12 月 2 日早期公開）



藤本　章良

令 2 大阪大学基礎工学部情報科学科卒．現在，同大学大学院情報科学研究科博士前期課程在学中．ソースコード分析に関する研究に従事．

肥後　芳樹　（正員）

2002 大阪大学基礎工学部情報科学科中退．2006 同大学大学院博士後期課程了．2007 同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助教．2015 同准教授．博士（情報科学）．ソースコード分析，特にコードクローン分析，リファクタリング支援，ソフトウェアリポジトリマイニング及び自動プログラム修正に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．

松本淳之介

平 30 大阪大学基礎工学部情報科学科卒．令 2 同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博士前期課程了．在学時リポジトリマイニングに関する研究に従事．

楠本　真二　（正員）

1988 大阪大学基礎工学部卒．1991 同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 同講師．1999 同助教授．2002 同大学大学院情報科学研究科助教授．2005 同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．情報処理学会，IEEE，IFPUG 各会員．