特別研究報告

題目

抽象構文木を利用したファイル間の移動コード検出

指導教員

楠本 真二 教授

報告者

藤本 章良

令和 2 年 2 月 10 日

大阪大学 基礎工学部 情報科学科

令和元年度 特別研究報告

抽象構文木を利用したファイル間の移動コード検出

藤本 章良

内容梗概

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．ソースコードの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コードを入力として受け取ると，内部で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という問題点がある．そこで本研究では，プロジェクトに含まれる全てのソースファイルから 1 つの抽象構文木を構築し，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．そして 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，全てのプロジェクトから合計で 76,600 個のファイルを横断する移動を検出できた．また，検出結果を確認したところ，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，いくつかの特徴が得られた．さらに，ソースコードの移動を検出できる既存ツールと比較を行った結果，提案手法のみが検出できる移動の特徴についても明らかになった．

主な用語

差分，GumTree，抽象構文木

目次

図目次

表目次

1 はじめに

ソフトウェア開発において，バージョン管理システムの使用は必須であり，多くのプロジェクトで導入されている [1]．ソフトウェア開発において，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，ソースコードへの理解が深まり，振る舞い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, 6, 7] が開発されている．バージョン管理ツールのひとつである `Git` では，その内部に Myers や Histogram などの差分検出アルゴリズムを選択可能な `diff` コマンドが組み込まている [8]．

しかし，Unix の `diff` に代表されるテキストベースの差分検出ツールには 2 つの問題点が存在する．1 つ目の問題点は，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して出力する．また，文を `if` 文や `for` 文のブロック内に移動させる，またはブロックの外へ移動させた場合，そのブロック全体が変更されたと出力する．このため，どの部分に対して実際に変更が行われたかが分かりづらくなる [5, 9]．`diff` の出力結果に基づいて，より詳細に変更箇所をハイライトするツール[*1] も存在するが，後者の例ではハイライトが行われない．2 つ目の問題点は，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で出力されるため，それ以外の操作を行った場合，開発者が意図した差分が出力されない可能性がある．

これらの問題点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, 10, 11, 12] が開発されている．そのひとつとして，GumTree [11] がある．GumTree は，差分を理解したい変更についてその前後のソースファイルを入力として受け取ると，その内部でそれぞれの 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るツールである．AST ノード単位の比較は，行単位での比較よりも細かい粒度で比較可能であるため，`diff` に比べてより適切な範囲で差分を出力可能である．さらにノードの削除や挿入以外にも，更新や移動を検出できる．

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 のビルドファイルの解析 [13] やバグ修正パターン検出の自動化 [14]，バージョン管理システムを利用する際のコミットメッセージの生成 [15]，API のコードのサジェスト [16] などに用いられている．

しかし，GumTree にも問題点が存在する．各ファイルに対して個別に計算を行うため，ファイルを

[*1] diff-highlight．`https://github.com/git/git/tree/master/contrib/diff-highlight`

横断するソースコードの移動を検出できない点である．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 [17].「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持つ可能性がある．

そこで，単一ファイル内の差分しか計算できない GumTree を拡張して，プロジェクト全体の差分を計算し，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．そのための手法として，複数のソースファイルからそれぞれ AST を生成し，それらを 1 つにまとめたプロジェクト全体の AST を構築する．変更前後のプロジェクト全体の AST を比較し，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．

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，合計で 76,600 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検証を行った．また，ソースコードの移動を検出できる既存のツールと比較し，検出されるファイルを横断するソースコードの移動の数や，そのソースコードの特徴を分析した．さらに，提案手法によって検出された，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．

以降，2 章では準備として AST や GumTree について説明を行う．3 章では研究動機として，GumTree の問題点を述べ，4 章では提案手法について説明する．5 章では，本研究で設定したリサーチクエスチョンについての説明を行い，6 章でそのリサーチクエスチョンに答えるために行った実験の内容と結果を述べた上で，7 章で実験結果の考察を行う．8 章では本研究の妥当性の脅威について述べ，最後に 9 章で，本研究のまとめと今後の課題について述べる．

2 準備

2.1 抽象構文木（AST）

AST は，ソースコードを構文解析して得られる木構造のデータである．AST の各ノードは以下の 5 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，木構造上の親ノードへの参照を持つ．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，木構造上の子ノードへの参照を持つ．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： 各ノードが持つラベル以外の情報である．例えば，メソッド名や変数名がこれにあたる．

2.2 GumTree の差分検出

GumTree は入力として，変更前のソースファイルと変更後のソースファイルを受け取り，AST のノード単位の編集スクリプトを出力する．編集スクリプトとは，変更後のソースコードを得るために変更前のソースコードに適用された編集操作の列である．GumTree は，削除・挿入・移動・更新の 4 つの編集操作を検出可能である．操作の種類とそれが行われた AST のノードの情報を，編集スクリプトとして出力する．

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，この 2 つの AST の違いを編集スクリプトとして出力する．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．1 段階目のトップダウンフェーズでは，2 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．2 段階目の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マッチングを行う．トップダウンフェーズで対応付けられたノードの親の部分木について類似度を計算し，閾値を超えていれば対応付けを行う．さらにその子孫にあたるノードに対し，RTED のアルゴリズム [18] を用いてさらにマッチングできるノードがないかを探索する．これを根に向かって繰り返し，ボトムアップフェーズが完了する．

対応付けられたノードは，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，どのノードが削除・挿入されたか，どのノードの位置が変更されたか，

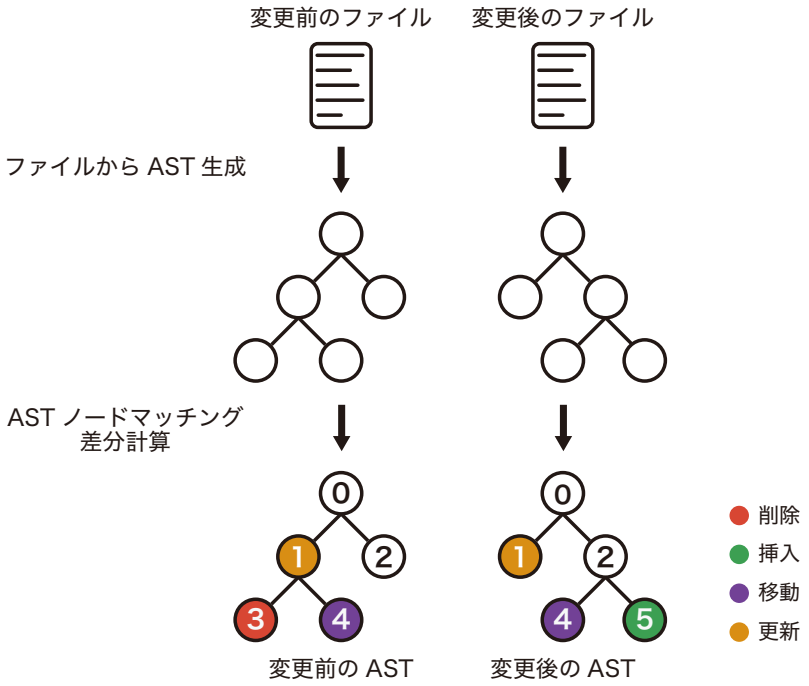


図 1　GumTree の差分検出

どのノードの値が更新されたか，どのノードは変更が行われていないか，といった情報を得る．

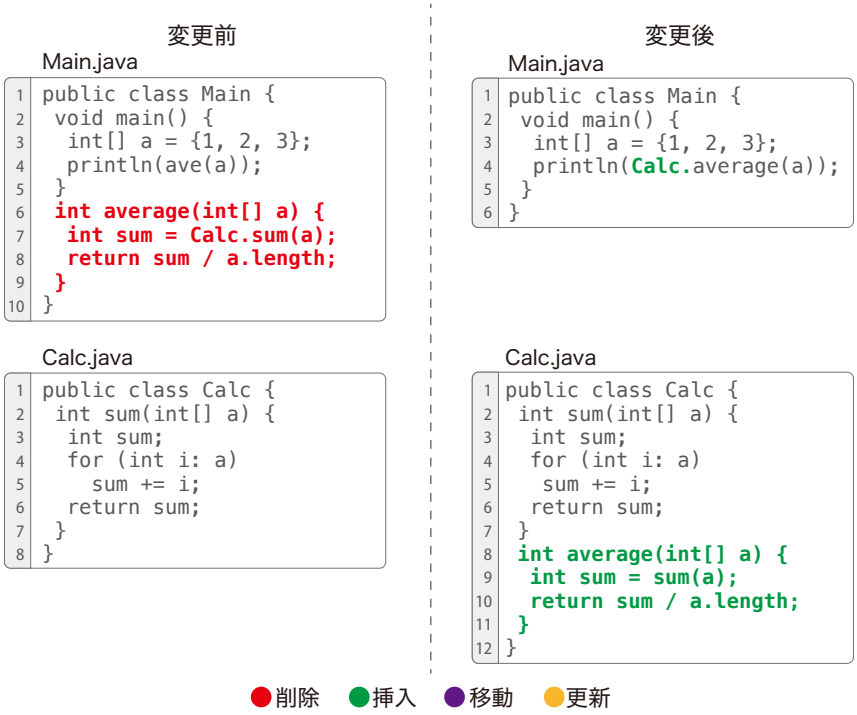
図 1 は，GumTree の差分検出方法を模式的に表している．変更前後のファイルから生成した AST にマッチングを行い，ノードの対応付けがされている．図 1 では，マッチングの結果を数字で表している．AST ノードの対応付けを基に，Chawathe ら [19] の変更前後の AST の木構造の違いを計算する手法を用いて，GumTree は差分を出力する．この例では，3 番のノードは変更前にしかないため削除，5 番のノードは変更後にしかないため挿入，と出力される．4 番のノードは変更前後のどちらにも存在しているが，親のノードが変わったため移動と出力される．1 番のノードは，更新と出力されている．更新は，変更の前後に同じノードが存在し，その値が変更された場合に出力される．

3 研究目的

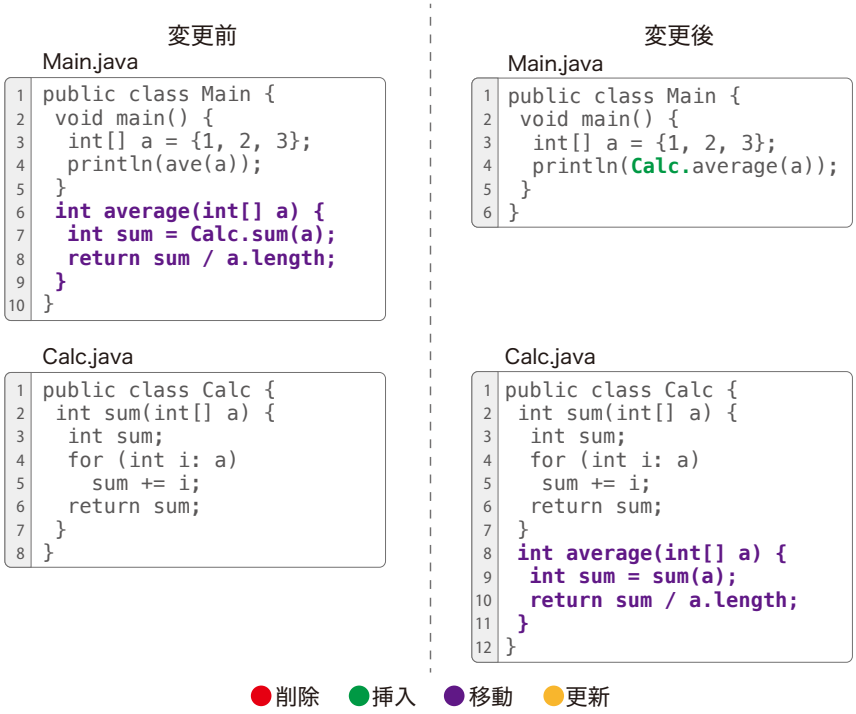
GumTree は，単一ファイルの変更について，削除・挿入・移動・更新のうちどのような操作が行われたかを検出できる．しかし，一度に複数のファイルに対して変更が行われた場合，ファイルを横断するソースコードの移動を検出できない．

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，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()` の移動の方が適切である（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持ち，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．

差分を表示する際，複数のファイルの変更を考慮することで，より適切な差分を出力できると著者は考えた．そこで，本研究では複数のファイルが変更された際に，ファイルを横断する移動を検出する手法を提案する．



(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

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，プロジェクトに含まれる全ての変更前後のソースファイルであり，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ソースファイルから，それぞれに対応する 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 1 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 節で述べる．

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い，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 1 つの 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，移動を検出する精度が下がったりする可能性がある．そこで，4.2 節，4.3 節の工夫により，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 1 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，ソースファイル名の辞書順とする．

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，プロジェクト全体の AST のノード数が増え，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，無変更であるにも関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．変更があるソースファイルから生成した AST だけを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，そのファイルから生成される AST をプロジェクト全体の AST には加えない．

- 全く変更がない
- 空行・スペース・タブの削除，挿入のみ

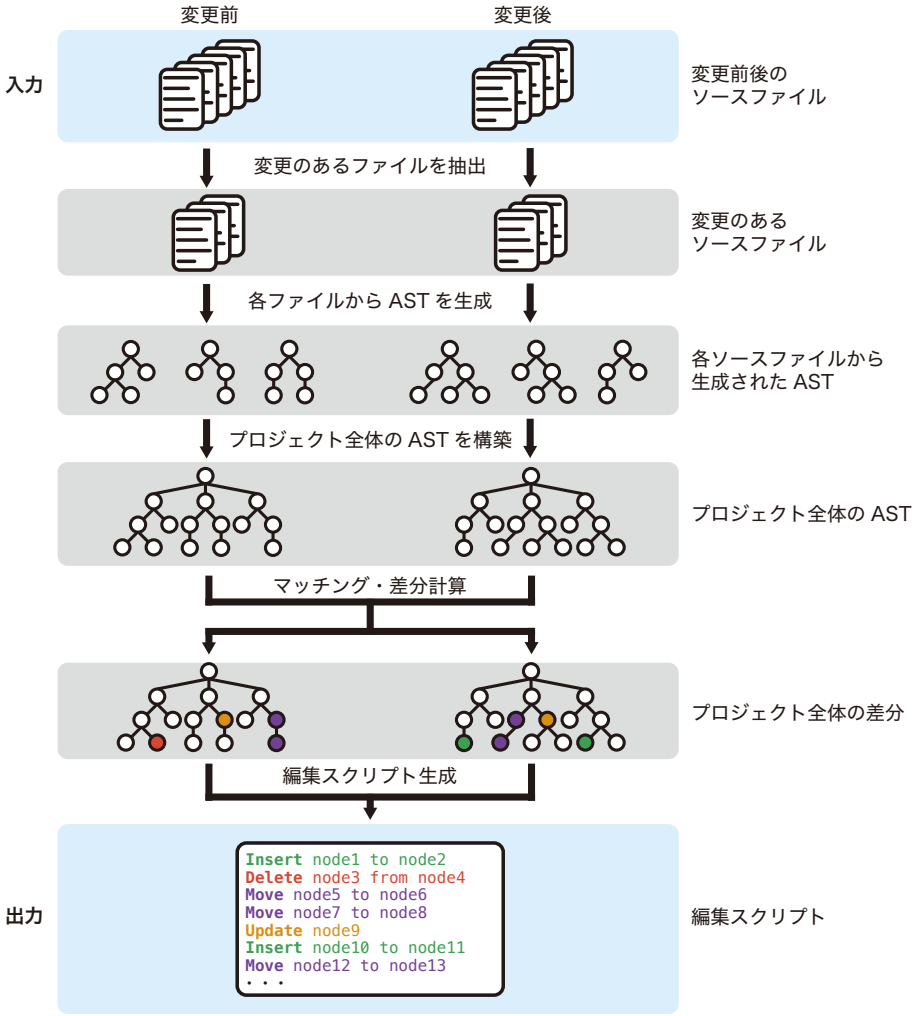


図 3　提案手法の概要

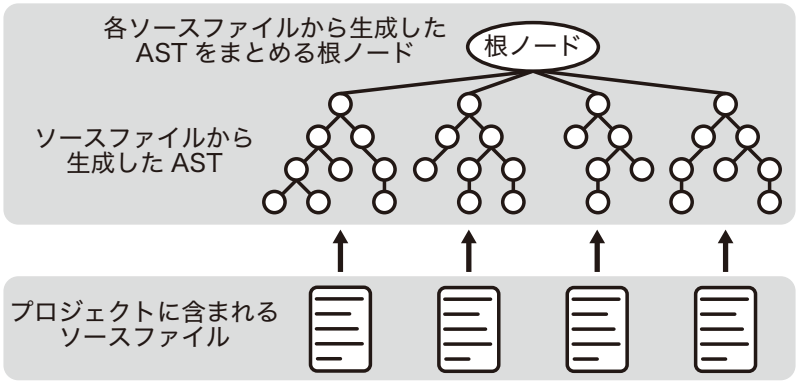


図 4　プロジェクト全体の AST の構築

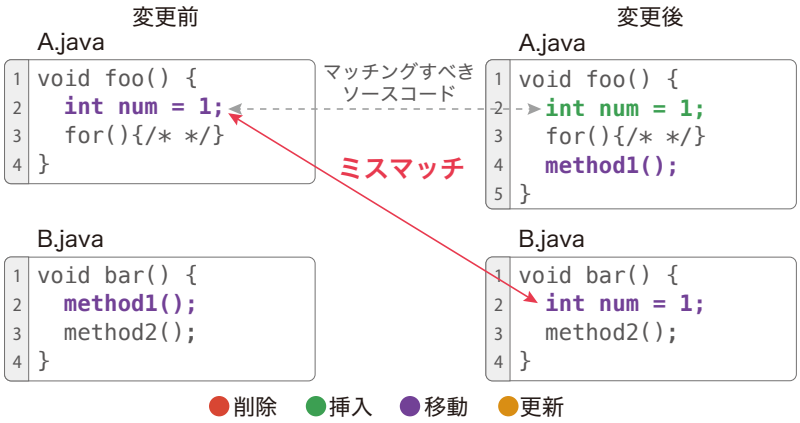


図 5　不適切なマッチングによる誤検出の例

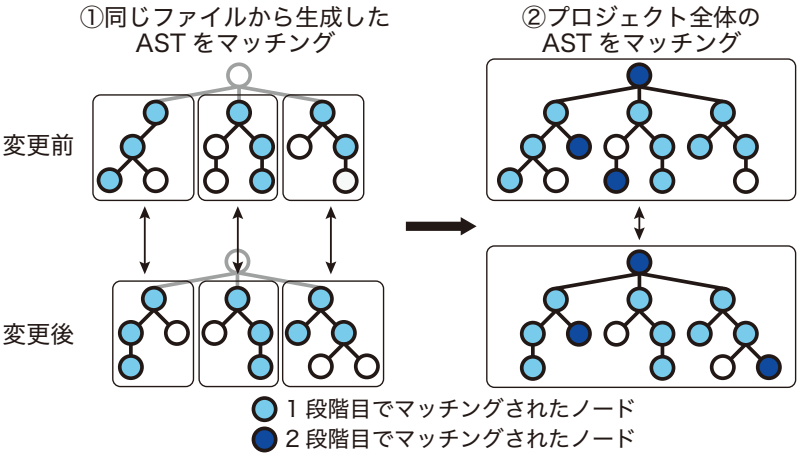
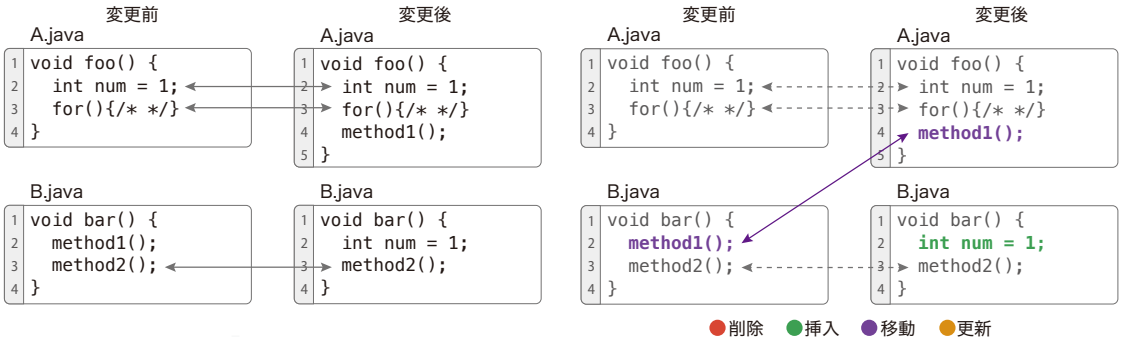


図 6　2 段階のマッチング

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際，変更されていないにも関わらず，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，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.java2` 行目の `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候補が増えるため，適切なノードを探し出せないからである．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 の中から，同じ



(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

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 につい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行う．これは AST ノードのマッチング候補を同一ファイルからのみにすることで，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 AST 間のノードをマッチングできる．2 段階目のマッチングでは，プロジェクト全体の AST において 1 段階目のマッチングでマッチングされていない AST ノードに，マッチングの候補が限定されている．これによって，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．

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が検出できる．

5 Research Question

本研究では，3 つのリサーチクエスチョン（以下 RQ）を設定し，これらを基に提案手法の評価や得られた結果の分析を行った．

RQ1: ファイルを横断するソースコードの移動をどの程度検出できるか．またそれらは差分の理解につながるか．

RQ2: ファイル間の移動コードを検出できる既存ツールと比較して，どのような違いがあるか．

RQ3: ファイルを横断するソースコードの移動にはどのような特徴があるか．

以下では，設定した各 RQ について説明する．

RQ1: ファイルを横断するソースコードの移動をどの程度検出できるか．またそれらは差分の理解につながるか

提案手法によってファイルを横断するソースコードが検出できるかどうかを検証する．さらに，検出した全ての移動に対するファイルを横断する移動の割合について調査し，提案手法の有用性を確認する．

また，提案手法によって検出できたファイルを横断する移動が，差分の理解につながるかを確かめる．提案手法で検出したファイルを横断する移動全てが，差分の理解につながる移動ばかりではない．差分の理解につながらないと考えられる移動の例を図 8 に示す．図 8 は Eclipse に対して提案手法を実行した際に，移動元と移動先のソースコードが大きく異なっているにも関わらず，ファイルを横断する移動であると出力された例である．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・削除といった複数の操作が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，この結果が差分の理解につながるとは考えにくい．他にも，定型処理が複数のファイルで個別に削除・挿入された場合，それらが無関係であるにも関わらずマッチングされる可能性がある．この結果，開発者が移動の意図を持たず行った操作が，移動であると検出される可能性もある．このような場合も差分の理解にはつながらない．

そこで，提案手法によって検出されたファイルを横断する移動のうち，差分の理解につながる移動がどのくらいの割合で存在するかを調査する．変更者が移動の意図を持って移動させたソースコードであることが読み取れれば，それは差分理解につながっていると著者は考えた．よって，検出したソースコードの内容や移動元と移動先のファイル内容から，移動の意図があるソースコードであるかを確認する．

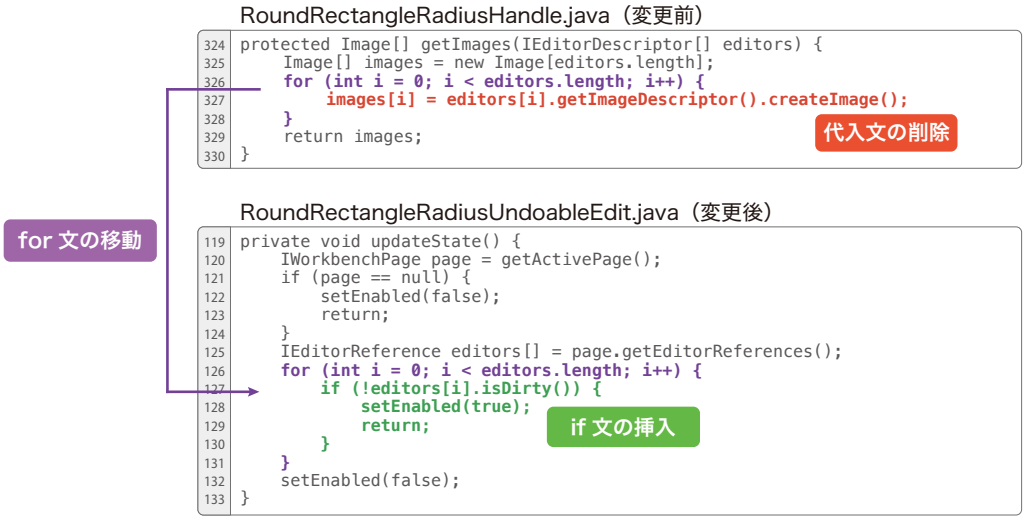


図 8　移動元と移動先のソースコードが大きく異なる例

RQ2: ファイル間の移動コードを検出できる既存ツールと比較して，どのような違いがあるか

リファクタリング検出ツール [20, 21, 22] 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．また，提案手法と既存ツールの両方から検出されたソースコード，あるいはどちらか片方のみで検出されたソースコードの特徴をそれぞれについて調べる．

RQ3: ファイルを横断するソースコードの移動にはどのような特徴があるか

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内容がどのように変化したか，移動前後のファイル名の特徴，などを調べる．

6 実験

5 章で設定した各 RQ への回答として，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，複数のオープンソースソフトウェア（以下，OSS という）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．

6.1 実験対象

GumTree の評価実験に用いられたデータセットである CVS-Vintage [23] 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．

6.2 実験方法

6.2.1 RQ1: ファイルを横断する移動の検出の確認

`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．

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 [*2]	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

[*2] 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．eclipse.jdt.core に対しても実験を行ったが，実行時間が長く評価できなかったため，実験対象から除外した．

次に，検出した移動が差分の理解につながるかどうかを確認する．この確認では，移動であると検出したコードを開発者が移動の意図を持って変更したかを判断するため，検出したソースコードの内容や移動元と移動先のファイル内容を目視で確認する．ただし目視確認をする際に，移動するノードを限定して確認する．限定するノードの種類は，移動と検出された AST ノードの部分木に含まれるノード数（以下，部分木の大きさという）によって決定した．部分木の大きさの平均が大きいノードに関して確認を行う．移動するノードの部分木が大きいと，移動するソースコードの量も多く，どのような機能が移動したかを開発者が理解しやすくなると考えられる．したがって，部分木が大きいノードの中で，差分の理解につながる移動を検出できるかの評価が重要となる．

提案手法によって検出された，ファイルを横断する移動を分析すると，移動する AST ノードのラベルの種類とその平均的な大きさには傾向があった．予備実験として，6.1 節に含まれる OSS のうち，ArgoUML と Apache Log4j 2，Apache Tomcat に対して提案手法を実行した際の，ファイルを横断して移動したノードの種類ごとの平均の大きさを表 2 に示す．ただし，ファイルを横断して移動したノードは種類が多いため，表 2 には一部のノードを抜粋して掲載する．

本研究では，予備実験を基に部分木の大きさが大きいノードの中から MethodDeclaration，WhileStatement，ForStatement，IfStatement，Block を選択し調査した．それぞれ，ソースコード中のメソッド，`while` 文，`for` 文，`if` 文，括弧で囲まれたブロックに該当する．目視確認では，ファイルを横断して移動したこれらのノードに対して，プロジェクトごとにそれぞれ最大 20 個をランダムに選択し確認を行う．

表 2　ファイルを横断して移動したノードをラベルで分類したときの平均の大きさ（一部抜粋）

ノードのラベル	ArgoUML	Apache Log4j 2	Apache Tomcat
MethodDeclaration	84.35	60.85	56.73
WhileStatement	160.54	92.33	102.47
ForStatement	111.25	62.00	55.50
IfStatement	120.94	26.66	32.42
Block	55.77	30.43	23.62
ImportDeclaration	2.00	2.00	2.00
FieldDeclaration	10.29	9.89	10.59
Assignment	6.56	7.76	6.32
SimpleName	1.00	1.00	1.00
全体	10.28	9.81	8.99

6.2.2 RQ2: 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

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 [20] 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，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 1 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．

次に，各手法が検出したメソッドの特徴を調査するため，メソッドが宣言されているクラスに関して調査を行う．メソッドの移動は，通常クラス間の移動の他に，内部クラスや匿名クラス間での移動が発生する可能性がある．これらの移動の数についても調査する．また，移動したのがメソッドかコンストラクタかであるかについても調査を行なった．

この実験は，6.1 項で挙げた 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 を実行する際は検出するブランチは 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．

6.2.3 RQ3: ファイルを横断する移動の特徴の確認

ファイルを横断するソースコードの移動の特徴を調査する．この時，どのようなソースコードが移動しているか，変更の前後でファイル内容がどのように変化したか，移動前後のファイル名の特徴，などを調べる．

この調査は，RQ1 の目視確認と同時に行う．調査するノードは，RQ1 と同じく MethodDeclaration，WhileStatement，ForStatement，IfStatement，Block の 5 つである．

6.3 実験結果

6.3.1 RQ1: ファイルを横断する移動の検出の確認

各 OSS とコミット数，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 3 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に対するファイルを横断する移動の割合である．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．しかし，その割合はプロジェクトによって 0.5%~10.9% とばらつきがある．

次に，部分木の大きさが大きいノードごとに移動した数を表 4 に示す．dnsjava の WhileStatement の移動を除いて，どのノードもファイルを横断して移動している．その数に注目すると，MethodDeclaration，Block が移動される回数が多く，その次に IfStatement が多い．WhileStatement や ForStatement の移動の数は他のノードに対して比較的少ない．また，WhileStatement と ForStatement の移動の数に大きな差がなく，同じような傾向が見られる．

検出したファイルを横断する移動のうち，差分の理解につながる移動の割合をプロジェクトごとに示したのが図 9 である．差分の理解につながる移動の割合は，全体で 76.4% であり，最も高い Apache Tocat では 93.4% であった．最も低い JHotDraw でも，51.5% であった．

差分の理解につながらないと判断した移動には，ソースコードの処理内容が移動前後で大きく異なっている以外に，同様のソースコードや定型処理が削除・挿入され，無関係なノード間でマッチングされた結果，移動であると検出された例が確認された．

表 3　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	9,514	(10.5%)	90,811
dnsjava	1,589	(0.5%)	325,362
Eclipse	25,177	(10.9%)	231,814
JHotDraw	5,711	(0.7%)	847,946
JUnit 4	2,830	(9.7%)	29,262
Apache Log4j 2	6,852	(5.8%)	117,102
Apache Struts	3,708	(7.2%)	51,768
Apache Tomcat	21,258	(8.9%)	259,447

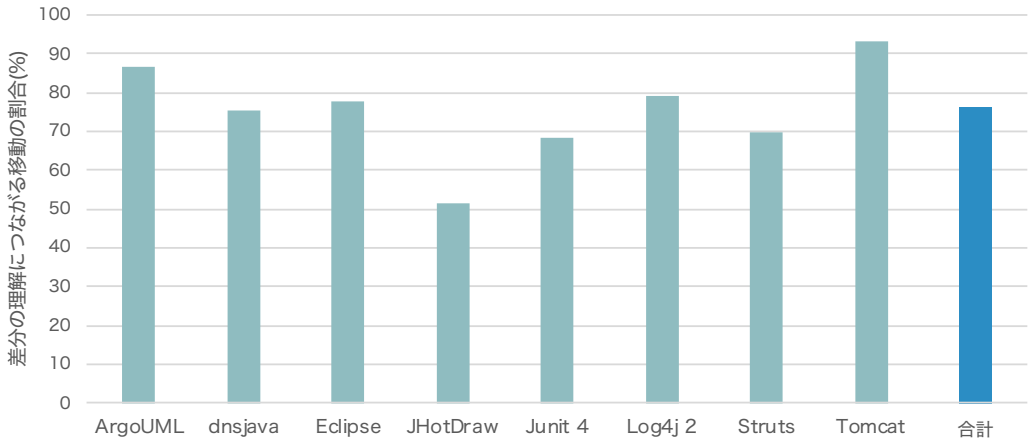


図 9　差分の理解につながる移動の割合

6.3.2　RQ2: 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

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 5 に示す．表 5 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．

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プロジェクトは 4 つで，移動の検出数においてどちらかの手法が優位であるとはいえない結果となった．また，提案手法と RefactoringMiner の両方が検出した移動の数は，それぞれの手法が検出した全ての移動の総数の 15%〜45% にとどまる．

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 6 に示す．匿名クラス

表 4　部分木の大きさが大きいノードごとの検出数

OSS 名	MethodDeclaration	WhileStatement	ForStatement	IfStatement	Block
ArgoUML	186	13	8	138	271
dnsjava	88	0	1	25	18
Eclipse	978	18	23	368	571
JHotDraw	204	1	7	48	149
JUnit 4	166	1	2	11	43
Apache Log4j 2	365	3	4	44	166
Apache Struts	211	1	2	43	97
Apache Tomcat	974	15	16	364	891

RoundRectangleRadiusHandle.java（変更前）

```
32  public class RoundRectangleRadiusUndoableEdit extends AbstractUndoableEdit {
        /* 省略 */
85      public void trackEnd(Point anchor, Point lead, int modifiersEx) {
            /* 省略 */
95          fireUndoableEditHappened(new AbstractUndoableEdit() {
112             @Override
113             public void undo() throws CannotUndoException {
114                 owner.willChange();
115                 owner.setArc(oldArc.x, oldArc.y);
116                 owner.changed();
117                 super.undo();
118             }
119         });
120     }
        /* 省略 */
125 }
```

RoundRectangleRadiusUndoableEdit.java（変更後）

```
26  public class RoundRectangleRadiusUndoableEdit extends AbstractUndoableEdit {
        /* 省略 */
53      @Override
54      public void undo() throws CannotUndoException {
55          owner.willChange();
56          owner.setArc(oldArc.x, oldArc.y);
57          owner.changed();
58          super.undo();
59      }
      /* 省略 */
84  }
```

図 10　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

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 10 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．

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．コンストラクタの移動の数の合計を表 7 にまとめる．コンストラクタの移動は，提案手法や RefactoringMiner どちらからも検出された．しかし，その数は提案手法の方が多いという結果が得られた．

表 5　提案手法と RefactoringMiner によって検出したファイルを横断するメソッドの移動数

OSS 名	両手法が検出した数	提案手法のみが検出した数	RMiner[*3]のみが検出した数
ArgoUML	103	76	121
dnsjava	17	53	43
JHotDraw	47	141	101
JUnit 4	69	97	61
Apache Log4j 2	166	276	138
Apache Struts	125	109	147
Apache Tomcat	703	270	575

[*3] RefactoringMiner の略表記

6.3.3 RQ3: ファイルを横断する移動の特徴

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説明する．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時に，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 11 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．この他にも，似た機能を持つクラスを統合して 1 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 32 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 12 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際に，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 13 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

表 6　メソッドが宣言されたクラスによる分類

	両手法が検出	提案手法のみが検出	RMiner のみが検出
匿名クラスで宣言されたメソッドの移動	0	29	0
内部クラスで宣言されたメソッドの移動	36	72	40

表 7　コンストラクタの移動数

両手法が検出	提案手法のみが検出	RMiner のみが検出
4	91	10

変更前

Main.java

```
1 class Main {
2   void read() {/* */}
3   void write() {/* */}
4   void prepare() {/* */}
5   void execute() {/* */}
6 }
```

変更後

Main.java

```
1 class Main {
2   void prepare() {/* */}
3   void execute() {/* */}
4 }
```

IO.java

```
1 class IO {
2   void read() {/* */}
3   void write() {/* */}
4 }
```

図 11　機能の分割のソースコードの例

変更前

Main.java

```
1 void method() {
2   while (!isDone()) {
3     /* some codes */
4   }
5   doSomething();
6 }
```

変更後

Main.java

```
1 void method() {
2   extraction();
3   doSomething();
4 }
```

Sub.java

```
1 void extraction() {
2   while (!isDone()) {
3     /* some codes */
4   }
5 }
```

図 12　メソッドへの切り出しのソースコードの例

変更前

C.java

```
1 class C {
2   void methodA() {/* */}
3   void methodB() {/* */}
4 }
```

変更後

C.java

```
1 class C extends A {
2   void methodB() {/* */}
3 }
```

A.java

```
1 abstract class A {
2   void methodA() {/* */}
3 }
```

図 13　継承関係の変化のソースコードの例

という名称がつく例が多く確認された．

短いソースコードの移動

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際によく見られた．特に Tomcat では継承関係のあるクラス間で発生した 13 個の移動うち，10 個が getter または setter であった．

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．

7 考察

7.1 RQ1: ファイルを横断する移動の検出

実験結果より，提案手法によってファイルを横断するソースコードの移動を検出可能になることが分かった．部分木の大きいノードは，表 4 で示したようにノードの種類によって移動した回数のばらつきが大きい．特に WhileStatement や ForStatement が移動した回数は少ない．考えられる理由は，AST に存在するノード自体の数である．表 8 は ArgoUML，Log4j，Tomcat の実験対象の最新コミットから生成された AST の中で，`if` 文，`while` 文，`for` 文に対応するノードの個数を計測した結果である．`while` 文や `for` 文は `if` 文に比べてソースコード中の出現数が少ない．そのためファイルを横断する移動の割合も低くなり，検出数も少なくなると考えられる．

また，差分の理解につながる移動の割合は全体で約 75%，最も低いプロジェクトでも 50%，半数以上のプロジェクトで 70% 以上となっており，ファイルを横断するソースコードの移動の検出に対して，提案手法は有効であるといえる．

7.2 RQ2: 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

表 6 より，匿名クラスと通常クラス間でのメソッドの移動は提案手法でのみ検出できた．これは，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の解析を行う．RefactoringMiner は JDT[*4]の AST を用いて解析を行なっているが，ルールのひとつとして，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

表 8　プロジェクトに含まれる `while`，`for`，`if` 文に対応するノードの数

ノードのラベル	ArgoUML	Log4j	Tomcat
IfStatement	12,426	5,080	23,521
WhileStatement	512	199	626
ForStatement	424	626	1,556

[*4] Eclipse のプラグイン（`https://projects.eclipse.org/projects/eclipse.jdt`）

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．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，2.2 節で説明したように，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 6 のような結果となった．

提案手法でのみ検出された移動の中には，匿名クラス内で宣言されたメソッドやコンストラクタの移動が含まれていることが明らかとなった．一方で，RefactoringMiner のみが検出したメソッドの移動についても調査を行なったが，目立った特徴はなかった．移動したメソッドを提案手法が検出できなかったか，RefactoringMiner が検出した移動のうちいくつか誤検出が存在している可能性が考えられる．

7.3 RQ3: ファイルを横断する移動の特徴

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 [24, 25]．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，表 4 からは，MethodDeclaration の移動が WhileStatement に比べて多いという結果が得られた．ある機能を持つソースコードを他のファイルに移す際，`while` 文や `for` 文を単体で移動させる変更は考えにくい．メソッドが 1 つの機能を持っている場合が多く，クラスが持つ機能を分割する際はメソッドの移動が中心になると考えられる．そのため，表 4 のように，MethodDeclaration の移動の数が多くなっていると考えられる．目視確認を行なった際には，`for` 文や `while` 文を含んだメソッドが，ファイルを横断して移動していることを確認した．

6.3.3 項で挙げた，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際，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．さらに，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，閾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．

8　妥当性の脅威

本章では，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．

実験においては，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．

9 おわりに

本研究では，各ファイルに対して個別に計算を行う 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 76,600 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．さら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

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本真二教授に心より感謝申し上げます．

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝申し上げます．

本研究に関して，有益かつ的確なご助言を頂きました，柗本真佑助教に深く感謝申し上げます．

本研究を進めるにあたって，多大なるご助力およびご指導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博士前期課程 2 年の松本淳之介氏に深く感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に心より感謝致します．

本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます．

最後に，これまでの 22 年間様々な面で支えて頂き，励まして頂いた家族にも心より感謝いたします．

参考文献

[1] Brian De Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? In *ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pp. 36–39. IEEE, 2009.

[2] David Binkley. An empirical study of the effect of semantic differences on programmer comprehension. In *International Workshop on Program Comprehension*, pp. 97–106. IEEE, 2002.

[3] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. ClDiff: Generating concise linked code differences. In *ACM/IEEE International Conference on Automated Software Engineering*, p. 679–690, New York, NY, USA, 2018.

[4] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, Vol. 1, No. 1, pp. 251–266, 1986.

[5] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *IEEE International Conference on Software Engineering*, pp. 595–598. IEEE, 2009.

[6] Alex Loh and Miryung Kim. LSdiff: A program differencing tool to identify systematic structural differences. In *ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, p. 263–266, 2010.

[7] Zhenchang Xing and Eleni Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *IEEE/ACM International Conference on Automated Software Engineering*, p. 54–65, 2005.

[8] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. How different are different diff algorithms in Git? *Empirical Software Engineering*, Vol. 25, No. 1, pp. 790–823, 2020.

[9] 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, pp. 2349–2359, 2008.

[10] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change Distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, Vol. 33, No. 11, pp. 725–743, 2007.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, pp. 313–324, 2014.

[12] Georg Dotzler and Michael Philippsen. Move-optimized source code tree differencing. In

IEEE/ACM International Conference on Automated Software Engineering, pp. 660–671, 2016.

[13] Christian Macho, Shane McIntosh, and Martin Pinzger. Extracting build changes with builddiff. In *International Conference on Mining Software Repositories*, pp. 368–378, 2017.

[14] Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. Towards an automated approach for bug fix pattern detection, 2018.

[15] Md Salman Ahmed and Anika Tabassum. Automatic contextual commit message generation : A two-phase conversion approach. 2018.

[16] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In *International Symposium on Foundations of Software Engineering*, pp. 511–522. ACM, 2016.

[17] Birgit Geppert, Audris Mockus, and F Robler. Refactoring for changeability: A way to go? In *IEEE International Software Metrics Symposium*, pp. 10–pp. IEEE, 2005.

[18] Mateusz Pawlik and Nikolaus Augsten. RTED: A robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, Vol. 5, No. 4, p. 334–345, 2011.

[19] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, Vol. 25, No. 2, p. 493–504, 1996.

[20] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *International Conference on Software Engineering*, pp. 483–494, 2018.

[21] Darren C Atkinson and Todd King. Lightweight detection of program refactorings. In *Asia-Pacific Software Engineering Conference*, pp. 8–pp. IEEE, 2005.

[22] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Working Conference on Reverse Engineering*, pp. 263–274, 2006.

[23] Martin Monperrus and Matias Martinez. CVS-Vintage: A dataset of 14 cvs repositories of java software. 12 2012.

[24] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[25] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, 2004.