抽象構文木内の階層移動に着目したGumTreeアルゴリズムの拡張

山本 貴之[1,a)]　松下 誠[1,b)]　肥後 芳樹[1,c)]

概要：開発者がソースコードの差分を迅速にかつ正確に理解するために，GumTree などの抽象構文木を用いた差分検出手法が提案されている．しかし，GumTree が抽象構文木間の階層移動を検出した際に，その移動情報をソースコード上で確認すると，開発者が認識する差分情報と認識が乖離する場合がある．本研究では GumTree を拡張し，このような変更前後の抽象構文木で生じた階層移動をソースコード差分から削除する手法を提案する．これにより，開発者がソースコード差分をより理解しやすくなることを目指す．提案手法を既存の変更前後のソースコードの組に対して適用した結果，約 17%の組から階層移動を検出した．これは，全てのソースコードの組から検出した移動総数の内約 17%を占める．さらに，検出した階層移動を目視で分類したところ，12 種類の変更パターンに分類できた．10 人の被験者を対象にして差分理解に関する実験を行った結果，変更パターン 12 種類のうち 11 種類において，従来の GumTree で出力した差分より提案手法で出力した差分が理解しやすいことが分かった．

1. はじめに

ソフトウェア開発では，コードレビューやデバッグの際に変更前後のソースコードの差分を確認する作業に多くの時間を要する [1,2]．このため，開発者はソースコード間の差分を迅速にかつ正確に理解する必要がある．開発者はソースコード間の差分を確認するために Unix の diff コマンドや Git の diff サブコマンドなどのツールを使用する．これらはソースコードの行単位の差分を検出し，挿入と削除の 2 種類の編集作業として差分を出力する．ツールの内部では Myers のアルゴリズム [3] など様々なアルゴリズムが差分検出に使用されている．一方で，これらのツールが出力する行単位の差分情報には，構造上の意味を持たないこと [4] や差分の粒度が粗いこと [5,6]，編集操作の種類が少ないこと [7,8] などの問題点が知られている．

このような課題を解決するために，抽象構文木（以下，AST）を用いた差分検出手法 [7–10] が提案されている．中でも GumTree [7,8] は AST を用いたソースコード差分検出ツールとして最も知られたツールの一つである．GumTree は AST を用いて差分を検出するため，開発者はソースコードにおける構造的な変更を容易に理解できる．また，差分を 4 種類の編集作業（挿入・削除・更新・移動）で表すため，diff コマンドと比べてより実装時の編集操作に近い情報を出力から読み取ることができる．GumTree は変更前後の AST から差分を計算した後，その結果を編集スクリプトと呼ばれるノードの編集操作一覧として出力する．その後，編集スクリプトの内容をソースコード上にマッピングする機能（以下，ソースコード上に差分情報をマッピングした出力のことを Diff View と呼ぶ）によって，開発者は視覚的に差分を確認できる．

しかし，GumTree には次のような問題点が存在する．ソースコードでは変更が無いように見えるコード片において，AST の構造に変化が生じたことで GumTree が移動を検出し，結果的に Diff View に差分情報が出力されてしまう場合がある．AST から計算された差分情報とソースコードから読み取れる差分情報に乖離がある場合，開発者の正確な差分理解を妨げる原因となる．

そこで本研究では，変更前後の AST における階層移動に着目し，GumTree の差分検出結果と開発者がソースコード上で認識する差分情報に乖離が生じる移動を検出・削除する手法を提案する．具体的には，GumTree を拡張し，階層移動を検出するアルゴリズムを追加する．これにより，開発者に対しより理解しやすい差分を提供することを可能にする．

提案手法を約 2000 の変更前後のソースコードの組に対して適用したところ，17.0%の組から，差分認識に乖離が

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Suita, 565–0871, Japan
[a)] t-yamamt@ist.osaka-u.ac.jp
[b)] matusita@ist.osaka-u.ac.jp
[c)] higo@ist.osaka-u.ac.jp

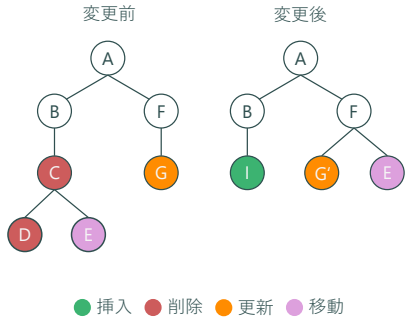

図 1 GumTree における変更前後の AST の差分検出

生じる可能性のある階層移動を検出した．また，検出した階層移動は総移動数の内 17.0%を占めていた．また，300 の変更前後のソースコードの組を対象に認識の乖離に繋がる階層移動を検出し，構造を分析したところ，12 種類の変更パターンに分類できた．これら 12 種類のパターンからそれぞれ 1 つずつ変更前後のソースコードの組を無作為に抽出し，コンピュータサイエンスを専攻する学生 10 人を対象に差分理解に関する被験者実験を実施した．その結果，11 種類の変更について既存手法よりも提案手法が出力した差分の方が理解しやすいという結果を得た．

2. GumTree

GumTree [7, 8] は AST を用いた差分検出ツールの中で最も有名なツールの一つである．GumTree は変更前後のソースファイルを入力とし，内部で AST に変換する．次に変更前後の AST において対応するノードを決定する．その後，ノードの対応関係を基に編集スクリプトと呼ばれるノードや部分木の操作列を出力する．最後に，編集スクリプト中のノード操作列をソースコード上にマッピングし差分を可視化する．

変更前後の AST に対する差分検出方法を図 1 を用いて述べる．ここで丸は AST のノードを表しており，ノードを識別するために英文字を付加している．まず変更前後の AST で対応するノードを決定する．図 1 ではノード A，B，E，F，G が対応している．変更前の AST に存在するノード C，D は対応するノードが変更後の AST に存在しないため，削除と判定する．変更後の AST に存在するノード I は対応するノードが変更前の AST に存在しないため，挿入と判定する．変更前の AST に存在するノード E は，変更後の AST において親ノードが F に変化しているため，移動と判定する．ノード G と G' は対応関係にあるが，値が異なるため更新と判定する．

実際にソースコード間の差分を検出し，差分情報を Diff View として表示した例を図 2 に示す．GumTree では編集スクリプトと Diff View の両方で差分情報を確認できるが，単なる AST 上のノードの操作列である編集スクリプトと比較すると，Diff View はより視覚的に差分を理解しやすい．

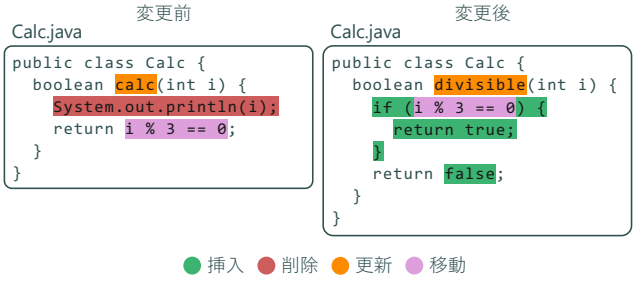

図 2 Diff View の例

図 3 開発者の差分認識例（提案手法の差分出力結果）

図 4 階層移動を検出した GumTree の出力結果

3. 研究動機

GumTree が検出した差分を Diff View で確認すると，開発者が想定していた差分情報と認識が乖離する場合がある．図 3 は Java の変更前と変更後のソースコードにおいて開発者が認識する差分の例である．return 文中の式に対し，論理式の演算子 `&&` と被演算子 `node.hasParent()` が追加されている．一方，GumTree は図 4 のように変化していない箇所 `node.isLeaf()` の移動を検出してしまう．このような差分情報を出力することは開発者の差分理解を妨げ，混乱を招く原因となる．

GumTree が図 4 のような移動を検出する理由は，変更前後の AST における階層移動にある．図 5 に，GumTree が図 4 の差分を検出する際に用いた AST を示す．図 5 では，ピンク色で示した MethodInvocation を根ノードとする部分木の親ノードが ReturnStatement から InfixExpression に変化し，階層が変化している．つまり，GumTree は AST における階層の変化を移動と判定する．しかし，図 3 のように，AST 上における階層移動が，Diff View 上でも移動しているように見えるとは限らない．本研究では，GumTree の差分検出結果とソースコード上での差分認識に乖離が生じる階層移動を検出する．検出した不要な階層移動を削除することで，図 3 のように開発者が Diff View 上で理解しやすい差分検出を目指す．

4. 提案手法

本節では 3 節で例に挙げた階層移動を検出し，削除す

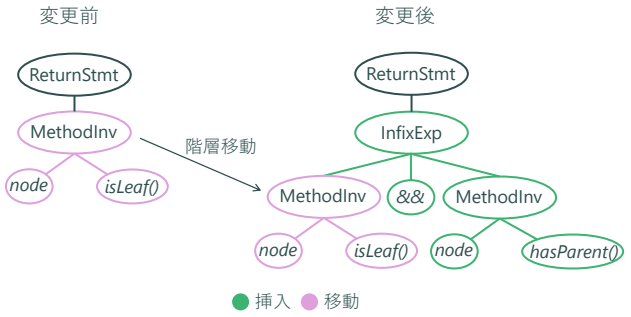

図 5 GumTree による変更前後の AST の差分検出結果

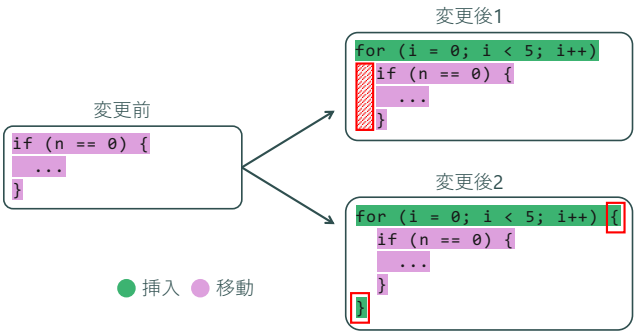

図 6 Statement の移動例

るアルゴリズムについて述べる．まず，提案するアルゴリズムに先立って階層移動の条件を定義する．その後，定義した条件を基に階層移動が生じる可能性のあるノードの調査を行う．最後に，調査したノードの情報を用いて階層移動判定アルゴリズムを提案する．なお本提案手法では，GumTree が標準で使用している Eclipse JDT Core[*1]（以下，JDT）が生成する AST を前提とする．また，対象となる Java のバージョンは最新の LTS である Java21[*2]とする．

4.1 階層移動が生じるノードの条件

本研究では，認識に乖離が生じる階層移動を，1) 同種類のノード内で完結する，2) 移動対象のノードが Statement より細かい粒度である，3) 変更前後の AST において横移動を含まない，の各条件を全て満たす移動と定義する．

1 つ目の条件として，本研究では同種類のノード内で生じる階層移動に着目する．異なる種類を跨ぐ移動はソースコードの構造が大きく変化するため，今まで通り移動と判定して構わないとした．例えば，MethodInvocation と InfixExpression はどちらも Expression という種類に分類され，図 5 における移動は同種類のノード内での移動となる．

2 つ目の条件として，本研究では Statement よりも粒度の細かいノードを対象にする．これは Block や Statement

[*1] https://github.com/eclipse-jdt/eclipse.jdt.core
[*2] https://docs.oracle.com/javase/specs/jls/se21/html/index.html

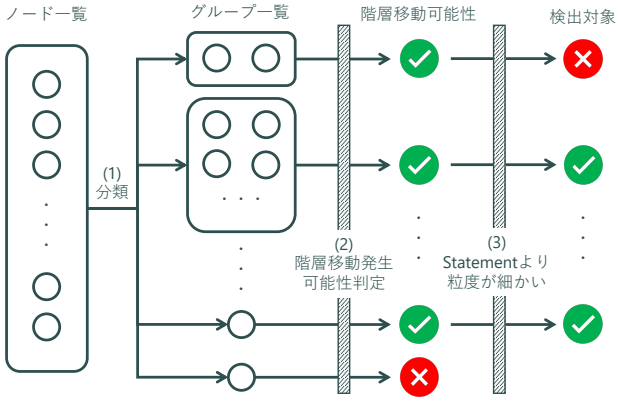

図 7 階層移動が生じる可能性のあるノードの調査手順

などの粗い粒度のノードにおける階層移動は，ソースコード上でも認知しやすいためである．図 6 は 2 種類の編集操作における Diff View であり，どちらも if 文が for 文のループの内側に移動している．どちらも Statement という同種類のノードにおける階層移動であり 1 つ目の条件を満たすが，インデントや中括弧 {} を使うことでソースコード上でも階層化を認識できる．これにより，AST 上の差分と Diff View 上の差分にあまり乖離は生じないと考えられる．一方で，図 4 のように Expression などの粒度が細かいノードにおいて階層移動が生じた場合，ソースコード上で階層情報を認識することが難しい．

3 つ目の条件における横移動とは，根ノードから移動の対象となる部分木までの経路が分岐している場合に生じた移動を指す．横移動を含む場合は Diff View 上でも明確にコード片の移動が認識できるため，階層移動ではなく従来の移動と判定する．一方，対象の部分木から根ノードまでの経路上にノードが追加・削除された場合，純粋な階層移動が生じる．純粋な階層移動が生じた場合，図 3，4 のように，単に他のコード片が挿入または削除されたように見えるため，差分情報として移動を出力する必要がない．

4.2 階層移動が生じるノードの調査

予め検出候補を絞るために 4.1 節で定義した条件 1, 2 を満たす可能性のあるノードを JDT の AST ノードの実装を基に調査した．図 7 は調査手順を表している．

まず JDT が持つ AST ノードクラスの継承関係を基にノードの分類を行う．例えば，ノードの MethodInvocation と InfixExpression を表すクラスは親クラスとして Expression 抽象クラス[*3]を持つため同グループに分類する．その結果，全 101 個の AST ノード[*4]を，複数の要素からなる 12 個のグループと単一の要素からなる 12 個のノードに分類した．ただし，全てのグループが排他的に構成されるわけで

[*3] https://github.com/eclipse-jdt/eclipse.jdt.core/blob/master/org.eclipse.jdt.core/dom/org/eclipse/jdt/core/dom/Expression.java
[*4] プログラムの構造に直接関係のないコメントや Javadoc を除く

はなく，グループ同士が包含関係になる場合もある．例えば，Expression グループは Annotation グループや Name グループを包含する．また，AST ノードは複数のグループに属す場合がある．Name グループが Expression グループに包含されることから，SimpleName は両方のグループに属すことになる．

次に，分類したグループ内で階層移動が生じる可能性の有無を判定する．各ノードに対して，そのノードが同じグループのノードを子ノードに持つ可能性を調査した．例えば，InfixExpression ノードは所属する Expression グループのノードを被演算子として子ノードに持つ．実際に図 5 では新しく挿入された InfixExpression が，同じく Expression グループに属する MethodInvocation を子ノードにすることで階層移動が生じている．グループの要素が単一の場合はそのノード自身を子ノードに持つ可能性を調査する．グループ毎に全ノードを確認したところ，同じグループのノードを子ノードに持つ可能性のあるノードの個数は表 1 のようになった．ただし，対象のノードが存在しないグループは表に記載していない．つまり，表 1 に記載した計 6 種類のグループにおいて階層移動が生じる可能性があると分かった．

抽出した 6 種類のうち，Statement よりも粒度が細かいのは Expression，Name，Pattern，Type である．よって，この 4 種類に対して階層移動を検出する．また，例外として if 文における階層移動も検出対象とする．これは if 文の else 節における階層移動が開発者の認識に反する場合があるからである．図 8 は，新しく else 節の if 文（`a < 0`）が追加されることで元の else 節（`a > 0`）が移動した例である．これは，if 文の構文的に，新しく else 節に条件文が挿入されたことで，元々あった else 節の階層が一段下がったからである．しかし，開発者にとっては単なる else 節の追加であり，既存の else 節の移動を認知しにくい．よって，if 文の else 節における階層移動も検出対象とする．

4.3 階層移動判定アルゴリズム

4.2 節で調査した情報を基に，階層移動を検出するアルゴリズムについて述べる．階層移動判定アルゴリズムでは，GumTree の差分検出手順に新しく階層移動の検出・削除の段階を追加する．変更前後の AST の差分を計算する従来のアルゴリズムや編集スクリプトからソースコードへの差分マッピング機能などはそのまま用いる．アルゴリズム 1 に階層移動判定アルゴリズムの概要を示す．

表 1 同じグループのノードを子ノードに持つノードの調査結果

グループ名	対象のノード数	全ノード数
BodyDeclaration	2	10
Statement	10	23
Expression	24	41
Name	1	3
Pattern	2	5
Type	6	9

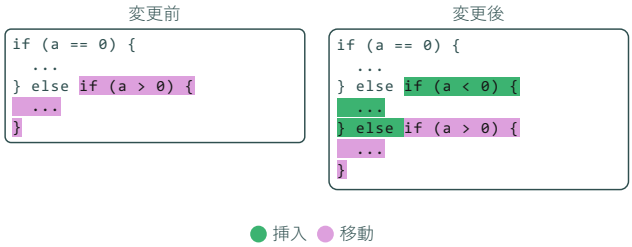

図 8 if 文の else 節で移動が検出される例

アルゴリズム 1: 階層移動判定アルゴリズム

Input: 変更前後の AST(t_1, t_2), GumTree が出力したノードの編集操作列 (E), GumTree が出力した変更前後の AST 間のマッピング集合 (M), 定義したノードの型集合 ($Types$)

```
1  for each e in E do
2      if e is not move action then
3          continue;
4      subTree_1 ← getSubTreeBeforeMove(e);
5      subTree_2 ← getSubTreeAfterMove(e);
6      if type(subTree_1) ∉ Types then
7          continue;
8      if !compareGrandParents(subTree_1, subTree_2)
         then
9          continue;
10     if !comparePaths(subTree_1, subTree_2) then
11         continue;
12     E ← E \ e;
```

まず，既存の GumTree のアルゴリズムを用いて AST 間の差分を計算し，編集スクリプトを出力する．その後，編集スクリプト中の移動操作のみに着目し（アルゴリズム 1，2-3 行目），4.1 節に挙げた 3 つの条件を満たすかどうかを判定する．まずは，移動した部分木の根ノードが 4.2 節で定義した 5 種類のノードに含まれるかどうか確認する（アルゴリズム 1，6-7 行目）．これにより，2 つ目の条件判定を行う．次に，後に示す 2 段階の条件判定を実施し，1 つ目と 3 つ目の条件を確認する．全ての条件を満たす移動の場合，編集スクリプトからその移動操作を削除する（アルゴリズム 1，12 行目）．

1 つ目の条件判定では，対象の移動が同種類のノード内で完結する移動であることを確認する．そのために，移動した部分木が変更前後で同じ先祖ノードを持っているかを判定する（アルゴリズム 1，8 行目）．ここで言う先祖ノードとは，移動した部分木の根ノードから全体の木の根ノードに向かって遡り，一番初めに到達した異なる種類のノードを指す．例えば，図 5 における変更前の AST の場合，MethodInvocation から根ノードに向かって遡ると，異なる

図 9 横移動が検出された変更前後のソースコード例

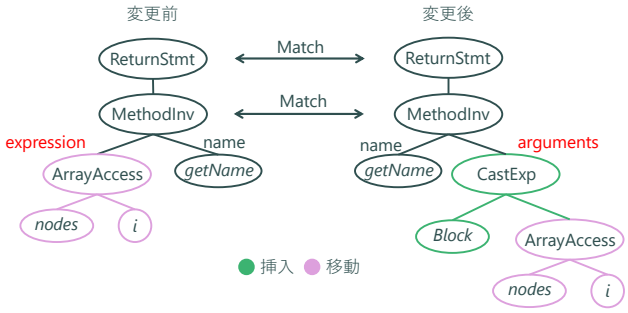

図 10 横移動が検出されたソースコードにおける変更前後の AST

種類である ReturnStatement に到達する．変更後の AST の場合，MethodInvocation から根ノードに向かって遡ると，同種類のノードである InfixExpression を通過し，異なる種類である ReturnStatement に到達する．それぞれ到達した ReturnStatement がマッチしている場合，先祖ノードが一致したと判定する．一方，先祖ノードがマッチしていない場合，異なる種類を跨いだ移動と判定する．

次に，3 つ目の条件判定では，対象の移動が横移動を含まないかを確認する（アルゴリズム 1，10 行目）．横移動判定では，図 9 に示す変更前後のソースコードにおける `nodes[i]` の様に明らかに移動と認識できる横移動を排除する．そのためには，1 つ目の条件判定で到達した先祖ノードから移動した部分木に至るまでの経路を比較し，経路に枝分かれがないかを確認する．

枝分かれは，変更前後の経路上に対応関係を持つノードを基準とする．経路上に対応関係を持つノードを基準に，次の経路上のノードが異なる子プロパティを持つ場合，横移動が発生したと判定する．子プロパティとは，構文上の子ノードとしての特性を表す情報を指し，根ノード以外の全てのノードが持つ．例えば，MethodInvocatiion はメソッド名の name やメソッド名の前に付く式の expression，引数を表す arguments などを子プロパティとして持つ．図 10 は図 9 のソースコードを AST で表している．図 10 では，変更前の経路上にあるノード ArrayAccess には expression が子プロパティとして付与されている一方，変更後の経路上にあるノード CastExpression には arguments が子プロパティとして付与されている．これにより，図 9 の移動は横移動を含むと判断し，3 つ目の条件に反することになる．子プロパティは JDT が AST を生成する際にノードに付与している情報を基にしている．従来の GumTree は差分検出の計算に子プロパティを使用していないため，提案手法で新たに AST のノードに情報を付与した．

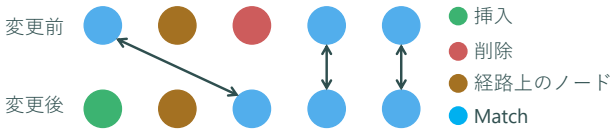

図 11 同じ子プロパティのノード列に対する対応ノードの LCS 結果

一方，同じ子プロパティを持つ場合でも枝分かれを判定できない場合がある．例えば，MethodInvocation は複数の引数を子ノードに持つことが可能であり，arguments という子プロパティが複数存在する場合がある．この場合，子プロパティが同じというだけで横移動がないと判断することはできない．そこで，同じ子プロパティを持つノード列に対して，対応関係のあるノードの最長共通部分列（以下，LCS）を使用した横移動判定を実施する．図 11 は同じ子プロパティを持つノード列において，対応関係のあるノードの対に対して LCS を計算した結果である．図 11 では LCS で得た対応関係を基準にすると経路上のノードは横移動したことになる．

5. 評価

提案手法を評価するために，4 つの Research Questions を用意した．

RQ1 本研究で定義した階層移動はどの程度存在するか？

RQ2 本研究で定義した階層移動には構造的にどのような変更パターンがあるか？

RQ3 本研究で定義した階層移動による差分を開発者は Diff View 上で理解できるか？

RQ4 提案手法を用いて出力した差分は開発者にとって理解しやすいか？

用意した 4 つの Research Question に回答するために，3 種類の方法 (自動評価，マニュアル評価，被験者評価) で評価を実施した．自動評価は RQ1 に，マニュアル評価は RQ2，被験者評価は RQ3 と RQ4 に対応している．

5.1 データセット

本評価では，GumTree [8] の評価で使用された Java のデータセットを使用する．データセットは変更前後のファイル 2045 組が含まれており，2 種類のデータセットで構成されている．1 つ目は Defects4J [11] というバグ修正が行われた変更の組のみを集めたデータセットである．2 つ目は GumTree [8] の評価時に著者らが独自に作成した GhJava というデータセットである．GhJava は有名な Java のプロジェクトから機能追加やリファクタリングなどによる変更を集めたデータセットである．GhJava はバグ修正が集められた Defects4J よりも複雑な変更を含んでおり，データが Defects4J が持つバグ修正に偏らないようにする目

的で収集された．これらのデータセットには JDT のパーサーによって構文解析に失敗するファイルが含まれておらず，本研究でも 2045 組の Java のソースコードは全て構文解析に成功した．2045 組の変更前後のファイルの組の内，Defects4J が 1046 組（530 commit），GhJava が 999 組（186 commit）で構成されている．

5.2 自動評価（RQ1）

データセットに含まれる全ての変更前後のファイルの組を対象に，階層移動を検出したファイルの組数とその割合，検出した階層移動数と総移動数に占める割合，検出された階層移動におけるノードの所属グループ毎の数（4.2 節で定義したノードの種類別分類）の 3 つの項目を測定した．

その結果，変更前後のファイル 2045 組において 348 組（17.0%）から階層移動が検出・削除された．検出された階層移動は，移動総数 3172 個の内 540 個（17.0%）を占めている．540 個の階層移動を 4.2 節で定義した 5 種類に分類すると，最も多いのは Expression グループであり，if 文の else 節，Name グループ，Type グループと続く．Pattern グループにおける階層移動は用意したデータセットからは検出されなかった．これは Pattern を使った実装が少ないことが影響していると考えられる．

5.3 マニュアル評価（RQ2）

300 組（Defects4J，GhJava それぞれ 150 組）の変更を対象に，検出した階層移動 106 個の変更パターンをより詳細に分類した．検出された階層移動は複雑な変更を含む場合があるため，ノードの種別とは異なり自動で評価することが難しい．よって，著者が全て目視で確認し分類を行った結果，変更パターンが 12 種類であることを確認した．表 2 は分類した 12 種類の概要と個数である．ただし，表中の個数と検出された移動が 1 対 1 に対応しているわけではないことに注意されたい．複数のパターンの要素を複合した階層移動も存在するため，1 つの階層移動につき複数のパターンの変更を測定することもある．また，4.1 節で定義した階層移動は，今回分類した 12 種類のパターン以外にも生じる可能性がある．

5.4 被験者評価（RQ3, RQ4）

被験者評価では以下の 2 つの項目を評価するために被験者実験を実施した．1 つ目は，開発者は従来の GumTree が検出した階層移動を Diff View 上で正確に理解できるかを評価する（RQ3）．2 つ目は，従来の GumTree と比べて提案手法を用いて出力した差分は開発者にとって理解しやすいかを評価する（RQ4）．

実験では，マニュアル評価で分類した 12 パターンにつき 1 つの変更差分をランダムに抽出し，既存手法と提案手法それぞれで検出した差分を被験者に提示した．その際，被験者に対して 2 つの質問を実施した．1 つ目の質問では，与えられた差分に関する説明，特に移動が判定されている場合にはどこからどこへ移動したのかの説明を求めた．2 つ目の質問では，与えられた差分はどちらの手法の方が理解しやすいかを 5 段階で選択してもらった．（1：提案手法の方が理解しやすい，2：提案手法の方が少し理解しやすい，3：どちらでもない，4：既存手法の方が少し理解しやすい，5：既存手法の方が理解しやすい）被験者はコンピュータサイエンスを専攻する学生 10 人であり，いずれも Java の経験が半年以上あり，基本的な構文を理解している．事前に GumTree が出力する差分に関する簡単な説明を実施し，疑問点がないことを確認した上で実験を開始した．

また，評価のバイアスを防ぐために実験時に次の 3 つの対策を施した．1 つ目は，先入観を与えないようにどちらの差分が提案手法かは被験者には伝えなかった．2 つ目は，手法の見せる順番によって差分の理解度に差が出ないように，被験者を 2 グループに分割し，先に見せる手法の順番をグループ毎に入れ替えた．3 つ目は，先に見たファイルペアの変更差分が後に見る差分の理解度に影響することを考慮し，被験者によって差分の見せる順番をランダムに入れ替えた．

5.4.1 RQ3

既存手法と提案手法それぞれの差分を見せたところ，既存手法が出力した 6 種類の階層移動において，被験者の過半数が移動の原因を正しく回答できなかった（表 3）．一方，提案手法が出力した Diff View では全てのパターンの差分について正確に説明できた．この結果は，3 節で述べた，AST で検出された階層移動を Diff View で確認した際に開発者の想定に反することを裏付ける．さらに，多くのパターンにおいて開発者はその移動の原因を理解できないため，本研究で定義した階層移動を Diff View に表示することは差分理解に悪影響を及ぼしていることがわかる．

5.4.2 RQ4

次に，既存手法と提案手法それぞれで出力した Diff View のどちらが理解しやすいか 5 段階でアンケートを取った結

表 2 マニュアル評価による階層移動のパターン分類結果

パターン名	パターン概要	個数
Exp-1	四則演算や論理演算での移動	41
Exp-2	メソッドの引数へ（から）の移動	34
Exp-3	メソッドチェーンの追加・削除による移動	15
Exp-4	丸括弧 `()` の追加・削除による移動	11
Exp-5	否定演算子 `!` の追加・削除による移動	6
Exp-6	キャスト演算子の追加・削除による移動	6
Exp-7	三項演算の要素へ（から）の移動	6
Name-1	ドット `.` で連結した名前の追加・削除による移動	4
Type-1	配列を表すブラケット `[]` の追加による移動	1
Type-2	他の型の要素型への移動	1
If-1	新しく追加された if 文の else 節に既存の if 文が連結したことによる移動	7
If-2	既存の if 文に対する else 節の追加・削除による移動	2

変更前
```
return
  TypeHandler.createValue(res, value);
```
変更後
```
return
  res == null ? null : TypeHandler.createValue(res, value);
```

図 12 既存手法が理解しやすいと判断された例（既存手法）

変更前
```
return
  TypeHandler.createValue(res, value);
```
変更後
```
return
  res == null ? null : TypeHandler.createValue(res, value);
```

図 13 既存手法が理解しやすいと判断された例（提案手法）

果を表 4 に示す．アンケートの結果，パターン Exp-7 を除く 11 パターンにおいて，平均値・中央値共に既存手法よりも提案手法の方が上回った．また，被験者毎に全てのパターンに対する回答の平均値と中央値を計算したところ，全ての被験者において既存手法よりも提案手法の方が上回っている．これらの結果は，本研究で検出・削除した階層移動のほとんどが Diff View 上では不要であり，開発者の差分理解を妨げる原因となることを示唆する．

唯一既存手法が検出した差分の方が理解しやすい結果となった，三項演算へ（から）の移動（Exp-7）における差分例を図 12，13 に示す．この例では，return 文中に三項演算が新しく挿入され，`TypeHandler.createValue(res, value)` が三項演算の要素に変化したことで階層移動が生じている．一方，提案手法ではこのコード片における階層移動を削除したため Diff View には表示していない．ただし，`res == null` は図中の return 文の外から移動してきたコード片であるため，階層移動とは無関係であることに注意したい．このような三項演算の要素に変化することによる階層移動は，被験者にとっては Diff View に表示した方が理解しやすいと考えられる．被験者にその理由を尋ねたところ，三項演算の要素に変化したことが一目瞭然であるため，既存手法による差分の方が理解しやすいという意見が得られた．

6. 妥当性への脅威

提案手法は GumTree の性能に依存している．GumTree が誤った検出結果を出力した場合は，その誤った差分情報の下で提案手法が適用されてしまう．

表 3 階層移動の原因を正確に理解した人数

パターン名	理解している人数	理解していない人数
Exp-1	3	7
Exp-2	9	1
Exp-3	3	7
Exp-4	6	4
Exp-5	6	4
Exp-6	4	6
Exp-7	9	1
Name-1	1	9
Type-1	3	7
Type-2	9	1
If-1	6	4
If-2	4	6

マニュアル評価では 300 組という限られた数の変更前後のファイルから検出された階層移動を分類したため，実際にはさらに多くの種類の階層移動が検出されると考えられる．さらに，被験者評価では，各パターン毎に代表する 1 つずつの差分についてしか実験できていない．よって，必ずしも同じパターンであれば同様の結果が得られるとは限らない．

また，本研究は対象のプログラミング言語を Java に，AST 生成ツールは JDT に限定している．他のプログラミング言語や AST 生成ツールで実施した場合に全く同じ手法が適用できるとは限らない．しかし，階層移動を検出し削除するという研究のアイデアや大まかな手法は他の言語やツールにも活用可能であると考えている．

7. 関連研究

これまで，AST を使用した構造上の差分検出手法は複数提案されてきた [7–10]．本研究の題材とした GumTree [7,8] は，Cobena らの手法 [12] を基に少ない計算量で移動や更新を検出可能なヒューリスティックな手法を提案し，OSS としても頻繁にメンテナンスされている優れた差分検出ツールである．GumTree は多くの研究の基礎となっており，様々な拡張が存在する．例えば，Higo ら [13] は GumTree を拡張することで，変更前後のファイルで生じたコピーアンドペーストを検出する手法を提案した．Fujimoto ら [14] は GumTree を拡張し，ファイルを跨いだ移動を検出する手法を提案した．Dotzler [15] らは移動に最適化したアルゴリズム MTDIFF を提案し，GumTree を含む複数の AST

表 4 提案手法と既存手法のアンケート結果

	5 段階評価						
パターン	1	2	3	4	5	平均値	中央値
Exp-1	10	0	0	0	0	1.00	1.0
Exp-2	3	2	4	1	0	2.30	2.5
Exp-3	10	0	0	0	0	1.00	1.0
Exp-4	7	3	0	0	0	1.30	1.0
Exp-5	6	2	1	0	1	1.80	1.0
Exp-6	8	2	0	0	0	1.20	1.0
Exp-7	2	2	0	2	4	3.40	4.0
Name-1	10	0	0	0	0	1.00	1.0
Type-1	8	2	0	0	0	1.20	1.0
Type-2	5	3	2	0	0	1.70	1.5
If-1	6	1	0	1	2	2.20	1.0
If-2	8	1	0	1	0	1.40	1.0

差分検出ツールにおける編集スクリプトを短くすることに成功した．

srcDiff [16] は AST を用いた差分検出ツールであるが，木構造の最適な編集差分をあえて目指さず，構文情報を基に開発者に理解しやすい差分を検出することを目指した．IJM [17] は GumTree や MTDIFF が精度の低い移動を検出することに着目し，Java 特有の構文情報を基に適切な差分を検出する手法を提案した．srcDiff と IJM は共に構文情報を基に差分情報を検出しており，その点では本研究の提案手法と類似している．しかし，AST における階層移動とソースコード上で確認できる差分情報の乖離に着目し，GumTree のエコシステム上で動作する点が異なっている．

8. おわりに

本研究では，GumTree を拡張することで，差分認識に乖離が生じる原因となる階層移動を検出し削除する手法を提案した．提案手法を既存の変更前後のソースコードに対して適用したところ，約 17%のソースコードからこのような階層移動を検出した．これらの階層移動は全ての移動の内約 17%を占めることが分かった．さらに，階層移動を目視で分類したところ 12 種類の変更パターンに分類可能であった．また，12 種類の変更パターンの内 11 種類において提案手法を用いて出力した差分の方が理解しやすいという結果を得た．今後は提案手法を他のプログラミング言語や AST 生成ツールに適用することが望まれる．

謝辞 本研究は JSPS 科研費 24H00692，21K18302，21H04877，23K24823，22K11985 の助成を受けたものです．

参考文献

[1] Baum, T., Schneider, K. and Bacchelli, A.: On the Optimal Order of Reading Source Code Changes for Review, *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 329–340 (online), DOI: 10.1109/ICSME.2017.28 (2017).

[2] Tao, Y., Dang, Y., Xie, T., Zhang, D. and Kim, S.: How do software engineers understand code changes? an exploratory study in industry, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/2393596.2393656 (2012).

[3] Myers, E.: AnO(ND) difference algorithm and its variations, *Algorithmica*, Vol. 1, No. 1, pp. 251–266 (1986).

[4] Maletic, J. and Collard, M.: Supporting source code difference analysis, *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 210–219 (online), DOI: 10.1109/ICSM.2004.1357805 (2004).

[5] Canfora, G., Cerulo, L. and Di Penta, M.: Ldiff: An enhanced line differencing tool, *2009 IEEE 31st International Conference on Software Engineering*, pp. 595–598 (online), DOI: 10.1109/ICSE.2009.5070564 (2009).

[6] 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. 2349–2359–2359 (2008).

[7] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M.: Fine-grained and accurate source code differencing, *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, New York, NY, USA, Association for Computing Machinery, p. 313–324 (online), DOI: 10.1145/2642937.2642982 (2014).

[8] Falleri, J.-R. and Martinez, M.: Fine-grained, accurate and scalable source differencing, *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3597503.3639148 (2024).

[9] Fluri, B., Wursch, M., PInzger, M. and Gall, H.: Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction, *IEEE Transactions on Software Engineering*, Vol. 33, No. 11, pp. 725–743 (online), DOI: 10.1109/TSE.2007.70731 (2007).

[10] Hashimoto, M. and Mori, A.: Diff/TS: A Tool for Fine-Grained Structural Change Analysis, *2008 15th Working Conference on Reverse Engineering*, pp. 279–288 (online), DOI: 10.1109/WCRE.2008.44 (2008).

[11] Jiang, J., Xiong, Y., Zhang, H., Gao, Q. and Chen, X.: Shaping program repair space with existing patches and similar code, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, New York, NY, USA, Association for Computing Machinery, p. 298–309 (online), DOI: 10.1145/3213846.3213871 (2018).

[12] Cobena, G., Abiteboul, S. and Marian, A.: Detecting changes in XML documents, *Proceedings 18th International Conference on Data Engineering*, pp. 41–52 (online), DOI: 10.1109/ICDE.2002.994696 (2002).

[13] Higo, Y., Ohtani, A. and Kusumoto, S.: Generating Simpler AST Edit Scripts by Considering Copy-and-Paste, *The 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE2017)*, pp. 532–542 (2017).

[14] Fujimoto, A., Higo, Y., Matsumoto, J. and Kusumoto, S.: Staged Tree Matching for Detecting Code Move across Files, *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, pp. 396–400 (2020).

[15] Dotzler, G. and Philippsen, M.: Move-optimized source code tree differencing, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, New York, NY, USA, Association for Computing Machinery, p. 660–671 (online), DOI: 10.1145/2970276.2970315 (2016).

[16] Decker, M. J., Collard, M. L., Volkert, L. G. and Maletic, J. I.: srcDiff: A syntactic differencing approach to improve the understandability of deltas, *Journal of Software: Evolution and Process*, Vol. 32, No. 4, p. e2226 (2020).

[17] Frick, V., Grassauer, T., Beck, F. and Pinzger, M.: Generating Accurate and Compact Edit Scripts Using Tree Differencing, *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 264–274 (online), DOI: 10.1109/ICSME.2018.00036 (2018).