修士学位論文

題目

変更前後の抽象構文木における階層移動に着目したソースコード差分の理解性向上

指導教員
肥後 芳樹 教授

報告者
山本 貴之

令和 7 年 1 月 28 日

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 ソフトウェア工学講座

令和 6 年度 修士学位論文

変更前後の抽象構文木における階層移動に着目したソースコード差分の理解性向上

山本 貴之

内容梗概

開発者が変更前後のソースコードの差分を迅速にかつ正確に理解することは，ソフトウェア開発において重要な作業である．開発者の差分理解を支援するために，自動でソースコード差分の検出を行う手法やツールが数多く提案されてきた．中でも GumTree は抽象構文木を用いた差分検出手法の中で最も知られているツールの一つであり，ソースコード間の構造上の差分を検出可能である．GumTree は変更前後の抽象構文木の差分を計算し，挿入・削除・移動・更新の 4 種類の差分情報をソースコード上にマッピングする．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乖離が生じる場合がある．

本研究ではこのような変更前後の抽象構文木で生じた階層移動に着目する．提案手法では，GumTree を拡張しソースコード差分の理解を妨げる原因となる階層移動を削除する手法を提案する．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ところ，12 種類の変更パターンの内 11 種類において，提案手法で出力した差分が従来の GumTree で出力した差分よりも理解しやすいことが分かった．

主な用語

ソースコード差分
抽象構文木
GumTree

目次

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ことが求められる．開発者はソースコード間の差分を確認するために様々なツールを使用する．例えば，Unix の diff コマンドやバージョン管理ツールである Git の diff サブコマンドなどが有名である．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．また，テキストファイルであればどのようなフォーマットでも差分を検出することが可能であり汎用性が高い．一方で，差分が構造上の意味を持たないこと [3] や差分の粒度が粗いこと [4, 5]，差分の編集操作の種類が少ないこと [6, 7] などが問題として知られている．

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 [6, 7, 8, 9] が提案されている．中でも GumTree[6, 7] 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ているツールの一つである．GumTree は AST を使っ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することが可能である．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差分から読み取れる．GumTree は変更前後の AST から差分を計算した後，編集スクリプトと呼ばれる AST 間におけるノードの編集操作一覧を出力する．その後，編集スクリプトを基にソースコード上に差分を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）を持っており，開発者は視覚的に差分を確認することが可能である．

しかし，GumTree には次のような問題点が存在する．GumTree が Diff View に出力した差分情報が開発者の認識と反する場合があり，開発者の混乱を招くことがある．例えば，ソースコード上では変更が無いように見えるコード片でも，AST の構造上移動が生じており，結果的に Diff View にも差分情報が出力されてしまうことがある．このように，ソースコードから読み取れる差分情報と AST から計算された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．

そこで本研究では特に変更前後の AST における階層移動に着目し，開発者が認識するソースコード上の差分情報と AST における差分検出結果に乖離が生じる移動を検出・削除する手法を提案する．提案手法では，GumTree を拡張し，階層移動判定アルゴリズムを追加する．階層移動判定アルゴリズムでは，GumTree が出力した編集スクリプトから Diff View に出力する必要のない階層移動を検出し，削除する．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，差分認識に乖離が生じる可能性のある階層移動を検出した．検出された階層移動は，全てのソースコードの組から検出された総移動数の内 17.0% を占めている．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．さらに，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．実験の結果，従来の GumTree と提案手法を比較したところ 11 種類の変更において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　準備

2.1　ソースコード間の差分検出

ソフトウェア開発の現場では開発者は変更前後のソースコードの差分を理解するのに多くの時間を要する [1, 2]．よって，ソースコードの差分を迅速にかつ正確に理解することが求められる．ソースコード間の差分を確認する用途は様々であるが，主にコードレビューやデバッグの際に使用される．

コードレビューでは，異なるバージョンのソースコードを見比べ，新たに追加された箇所や削除された箇所などを確認する．これにより，適切な変更が施されたかを把握する作業を行う．この作業は他人が書いたプログラムを確認する場合だけでなく，自分が実装した内容の成否を確認する場合にも行われる．デバッグでは，バグが混入したとされるソフトウェアのバージョンが分かっている際に，そのバージョン前後の差分を確認することで効率よくソースコード中のバグの箇所を特定できる．

このようなソースコード間の差分を確認する作業には様々なツールが使用されている．中でも Unix の diff コマンドや Git の diff サブコマンドが有名である．これらのツールは 2 つのソースファイルを入力とし，行単位での一致を判定し，行の挿入と削除で差分を表現する．これらのツールの内部では Myers のアルゴリズム [10] を初めとした様々なアルゴリズム [11] が差分検出に使用されている．

しかし，行単位の差分検出手法には主に 3 つの課題が存在する．1 つ目の課題は差分が構造上の文脈を持たないことである [3]．行単位の差分はただ行の変更が示されるだけであり，ソースコードにどのような構造的変更が為されたかは開発者が目で見て考えなければならない．プログラムの理解は脳に負荷がかかり時間を要す [12, 13] ため，構造的な情報を通して適切な差分を開発者に提供すべきである．2 つ目の課題は差分の粒度が粗いことである．行が完全に一致していない限り，その行は削除されて新しく他の行が追加されたと表現されるため，ある行の一部分が変更されただけでも差分として出力された行の中から実際に変更された箇所を探し出す必要がある [4, 5]．3 つ目の課題は差分の編集操作の種類が少ないことである [6, 7]．diff コマンドなどのツールは行の削除と挿入でしか差分を表現することができない．しかし，開発者はコード片を異なる場所に移動させたり，コピーアンドペースト [14] で同じ実装を使い回したりする場合がある．つまり，行単位の差分検出ツールは開発者が意図した編集操作を正しく表現することが困難である．

こういった行単位の差分検出手法における課題を解決するために，GumTree[6, 7] などの AST を用いた差分検出手法 [8, 9] が提案されている．GumTree は 2 つのソースファイルを入力とし，一度ソースコードを AST に変換した後に変更前後の AST で差分を計算する．その結果，変更前の AST から変更後の AST への編集操作列（挿入・削除・更新・移動）を出力する．GumTree については 2.3 節で詳しく紹介する．

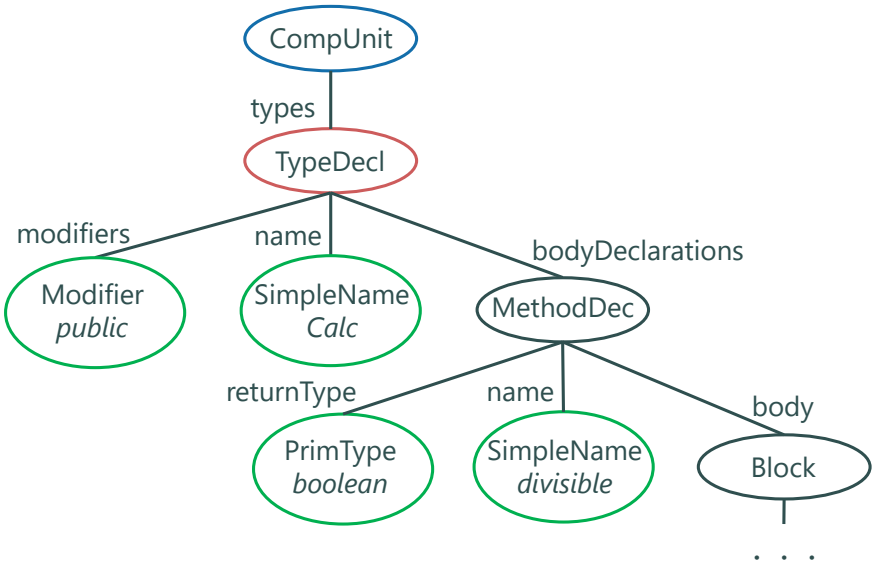

図 1　AST の例

2.2　抽象構文木（AST）

AST とはソースコードの構文構造を木構造で表したデータ構造である．ソースコードをコンパイルする際の中間的なデータ構造やプログラムを解析する際のデータ構造として広く使用される形式である．図 1 に AST の例を示し，AST で使用される用語の解説を行う．

根ノード AST 上の最上部（頂点）にあるノードを指す．図 1 では青色の CompilationUnit が根ノードである．

葉ノード AST 上の最下部（末端）にあるノードを指す．図 1 では緑色の 4 つのノードが葉ノードである．

親ノード あるノードにおいて頂点側に直接参照を持つノードを指す．図 1 では赤色の TypeDeclaration に対して青色の CompilationUnit が親ノードである．

子ノード あるノードにおいて末端側に直接参照を持つノードを指す．図 1 では青色の CompilationUnit に対して赤色の TypeDeclaration が子ノードである．

部分木 AST 中のあるノードを頂点とした木構造を指す．図 1 では MethodInvocation から下の全てのノードで構成される木構造のことを，MethodInvocation を頂点とする部分木と呼ぶ．

ラベル 構文上の特性を表す情報を指し，1 ノードにつき 1 つラベルを持つ．例えばクラス宣言や for 文などがある．図 1 では全てのノードがラベルを持つ（例：CompilationUnit，Modifier など）．

値 プログラム上で実際に使われているトークンの値を指し，1 葉ノードにつき 1 つ値を持つ．例えば変数名や型名などがある．図 1 では全ての葉ノードが値を持ち，斜体で表現されている

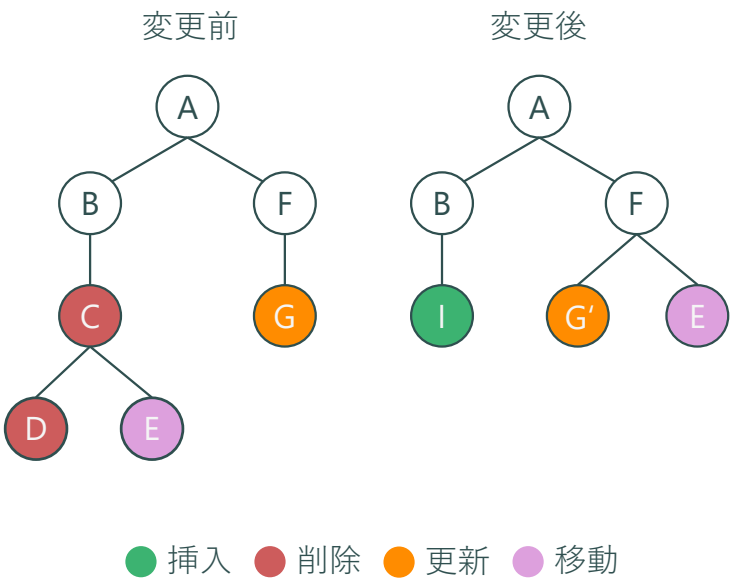

図 2　GumTree における変更前後の AST の差分検出

（例：public，Calc など）．

子プロパティ 構文上の子ノードとしての特性を表す情報を指し，根ノード以外のノード全てが持つ．図 1 では MethodDeclaration は 3 つの子ノードを持ち，それぞれのノードに対して returnType，name，body といった子プロパティが付けられている．

2.3　GumTree

GumTree[6, 7] は AST を用いた差分検出ツールの中で最も知られているツールの一つである．GumTree は 2 つのソースファイルを受け取るとそれぞれの AST の差分を計算し，最終的には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．その後，編集スクリプトの操作列を元にソースコード上にマッピングすることで差分を可視化することも可能である．

変更前後の AST の差分検出方法を図 2 を例に紹介する．まず，GumTree は変更前後の AST で互いに対応するノードペアを探す．図 2 の場合はノード A，B，E，F，G がマッチしたと判定する．マッチしなかったノードは，挿入もしくは削除されたと判定する．変更前の AST 上に存在するノード C，D は対応するノードが存在しないため削除されたと判定する．また，変更後の AST 上に存在するノード I も対応するノードが存在しないため新たに挿入されたと判定する．次に，対応するペアが見つかったノードの中で親ノードが変化した場合は移動と判定する．図 2 ではノード E の親がノード C からノード F に変化しているため移動と判定する．また，対応するペアが見つかったノードの中でラベルは同じだが値が異なる場合は更新と判定する．図 2 ではノード G が更新に該当する．

次に，実際にソースコードの差分を検出し，差分情報を Diff View として表示した例を図 3 に示

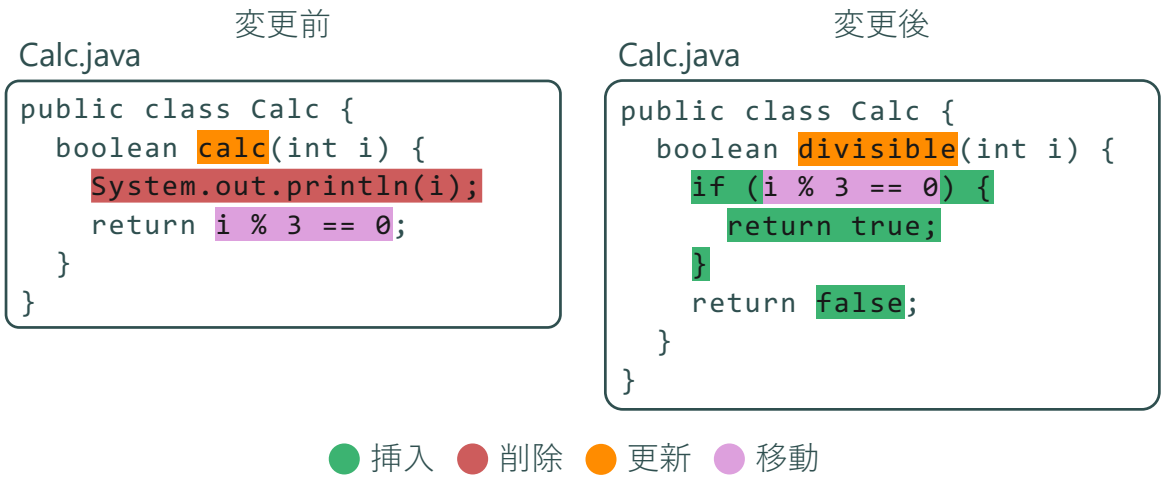

図 3　Diff View の例

す．GumTree では編集スクリプトか Diff View のいずれかを用いて差分情報を確認することが可能である．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．

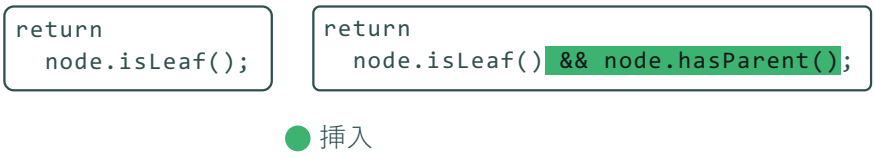

図 4　GumTree の検出結果と開発者の認識が異なる例（開発者の差分認識）

図 5　GumTree の検出結果と開発者の認識が異なる例（GumTree の検出結果）

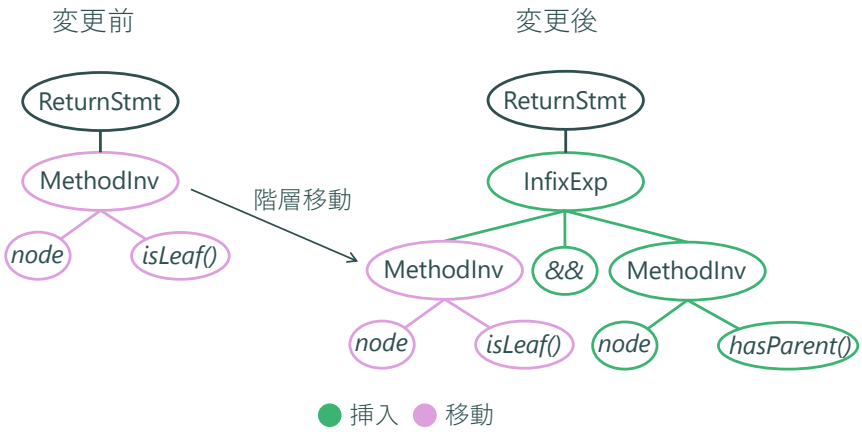

図 6　GumTree による変更前後の AST の差分検出結果

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 4 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 5 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．このような移動の検出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 5 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 6 に GumTree が変更前後の AST の差分を検出した結果を示す．GumTree は対応関係のある部分木の親ノードが変化するとその部分木は移動したと判定する．図 6 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 に変化している．従って，AST 上で部分木の階層移動が生じると部分木の親ノードが変化することになる．しかし，図 4 のように，AST 上で階層が移動しているからといって必ずしも Diff View

上で移動しているように見えるわけではない．本研究ではこのような問題が生じる移動を検出し，敢えて移動情報を無視した差分を出力することでソースコード差分の可視性向上を目指す．

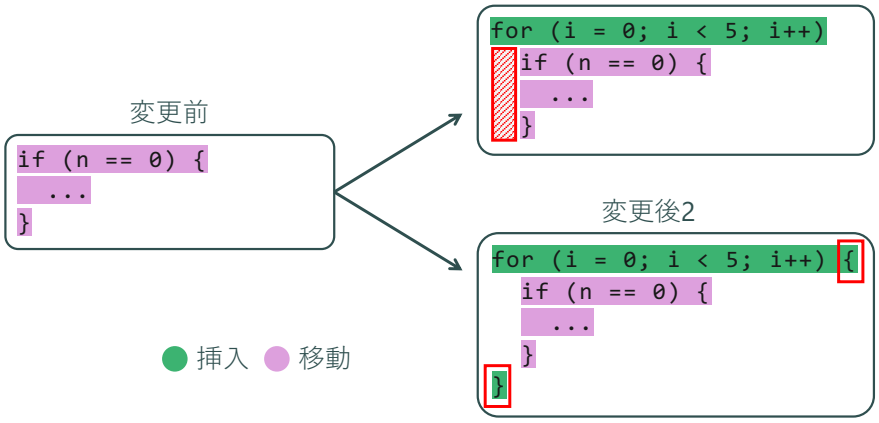

図 7　Statement の移動

4　提案手法

本節では 3 節で述べた階層移動を検出し，削除するアルゴリズムについて述べる．提案したアルゴリズムに先立って，まず階層移動が生じる条件を定義する．その後，定義した条件を基に階層移動が起こりう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 がデフォルトで使用する AST 生成ツールである 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次の条件を全て満たす移動と定義する．

1. 同種類のノード内で完結する移動である．
2. 移動対象のノードの粒度が Statement より細かいノードである．
3. 変更前後の AST において横移動を含まない上下の移動である．

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．複数の種類を跨ぐ移動はソースコードの構造が大きく変化しているため，今まで通り移動と判定して構わないものとする．例えば，MethodInvocation と InfixExpression はどちらも Expression という種類に分類され，図 6 における MethodInvocation 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における階層移動のみを対象とする．これは Block や Statement 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 7 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．2 つの編集操作の異なる点は，移動した if 文を

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

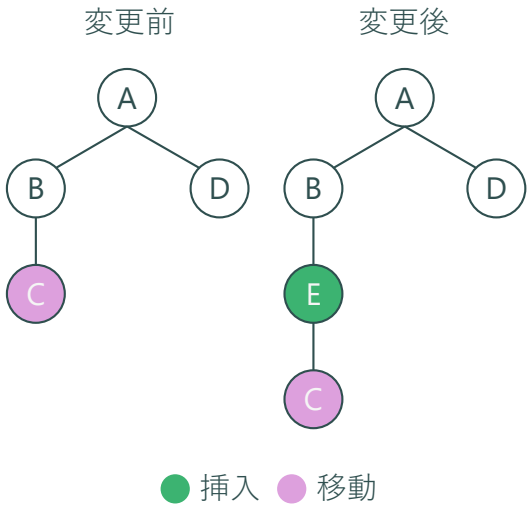

図 8　上下の移動例

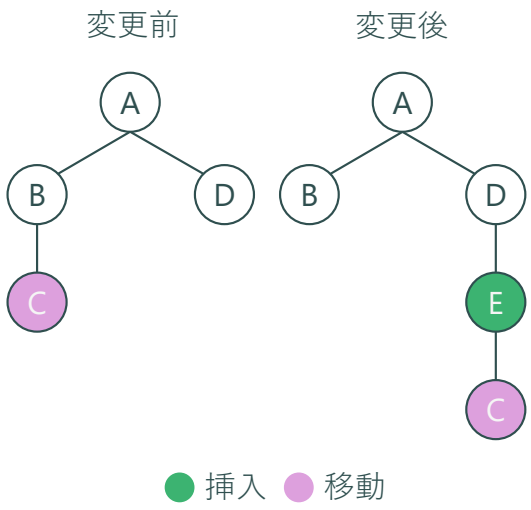

図 9　横の移動例

中括弧 {} で囲っているかどうかである．どちらも AST 上では Statement という同種類のノード内における階層移動となるが，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することが可能である．これにより，AST 上の差分と Web Diff 上での差分に乖離が生じることは少ないと考えられる．一方で，図 5 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判断することが難しい．

3 つ目の条件である変更前後の AST における上下移動とは，対象となる部分木から根ノードまでの経路上に新しくノードが追加，もしくは削除される場合に生じた移動を指す．図 8 の場合，ノード C から根ノード A までの経路上にノード E が追加されたことになり，C が下に移動したことになる．変更前後の AST で上下の移動が生じる場合，図 4，5 のように，Diff View 上で移動と判定されるコード片の前後に他のコード片が挿入または削除されたように見える．一方で変更前後の AST における横移動とは，対象となる部分木から根ノード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 Web Diff 上でも明確にコード片の横移動が判定できるため，階層移動ではなく従来の移動と判定する．図 9 の場合，新しく挿入されたノード E を除いたノード C からノード A までの経路が変更前と変更後で異なるため横移動と判定する．

4.2　階層移動が生じるノードの調査

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 10 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 MethodInvocation と InfixExpression を表すクラスは親クラスとして Expression 抽象クラス[*3]

[*3] https://github.com/eclipse-jdt/eclipse.jdt.core/blob/master/org.eclipse.jdt.core/dom/org/eclipse/jdt/core/dom/Expression.java

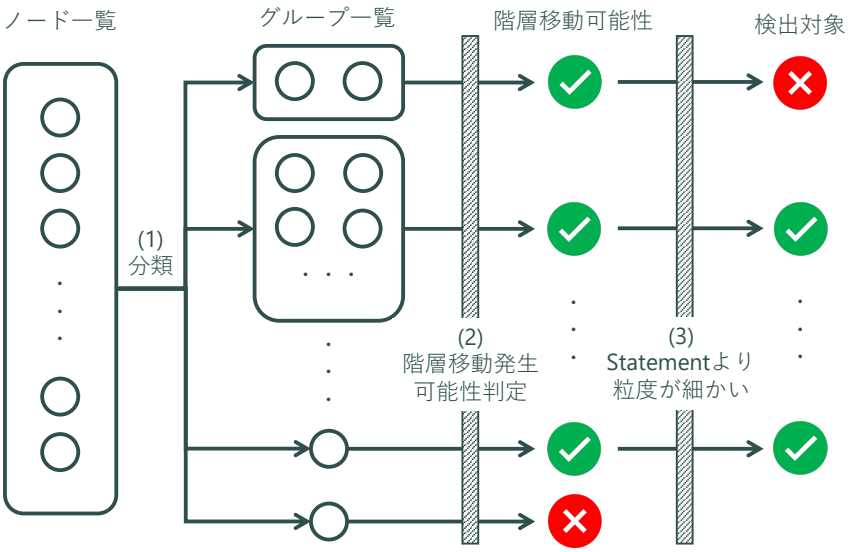

図 10　階層移動が生じる可能性のあるノードの調査手順

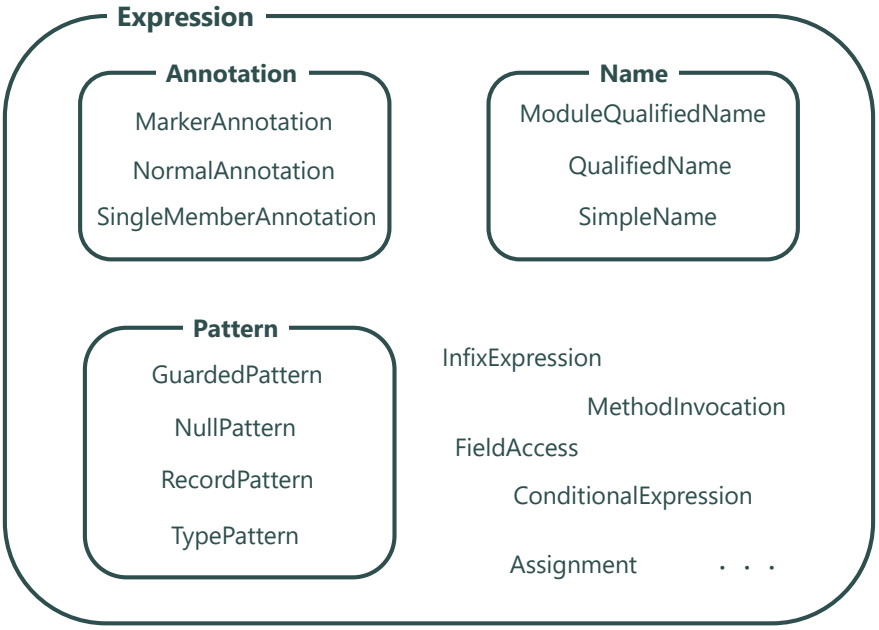

図 11　Expression グループ内の包含関係

を持つため同グループに分類する．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（詳細は付録 A を参照）．ただし，全てのグループが排他的に構成されるわけではなく，グループ同士が包含関係になる場合もある．例えば，図 11 では Expression グループが Annotation グループや Name グループを包含していることを

[*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

表している．また，AST ノードは複数のグループに属すことが可能である．図 11 の SimpleName は Name と Expression 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があるかを判定する．各ノードに対して，そのノードが所属グループのクラスを子ノードに持つかどうかを調査した．例えば，InfixExpression ノードは所属グループである Expression 抽象クラスのノードを被演算子として子ノードに持つため，Expression グループに属する他のノードを子ノードにすることができる．実際に図 6 では新しく挿入された InfixExpression が，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かで判定する．グループ毎にそのグループに所属する全ノードを確認したところ，子ノードに自分自身が所属するグループのクラスを持つノードは表 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ことが分かった．

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 文内における階層移動も検出対象とする．これは if 文における else 節の構文が開発者の認識に反する場合があるからである．図 12 は元々存在する else 節（`a > 0`）に変化がないのに移動が出力されている例である．これは，移動と判定された else 節の if 文の親ノードが，新しく挿入された else 節（`a < 0`）に変化したからである．しかし，開発者はこの変更を条件の追加と見なし，既存の else 節の移動を認知しない．よって，if 文内の else 節における階層移動も検出対象とする．

表 1　各グループにおける所属グループを子ノードに持つノード数

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

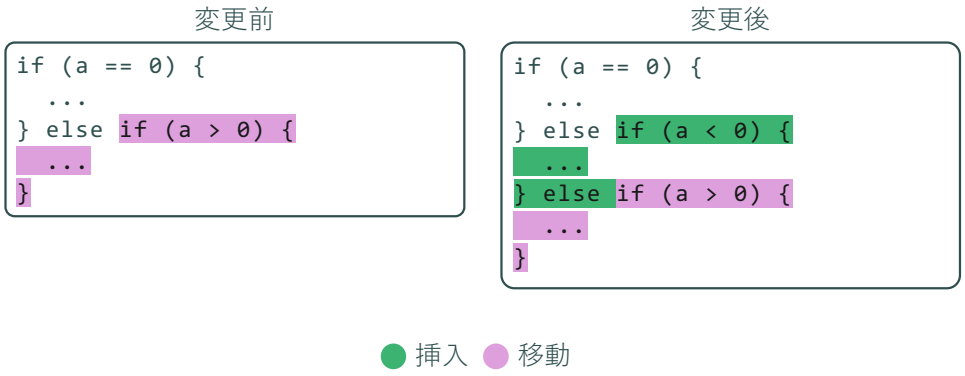

図 12　if 文の else 節で移動が検出される例

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 ($Types$)

```
for each e in E do
    if e is not move action then
        continue;
    subTree_1 ← getSubTreeBeforeMove(e);
    subTree_2 ← getSubTreeAfterMove(e);
    if type(subTree_1) ∉ Types then
        continue;
    if !compareParentsByBottomUp(subTree_1, subTree_2) then
        continue;
    if !comparePathsByTopDown(subTree_1, subTree_2) then
        continue;
    E ← E \ e;
```

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 のアルゴリズムに新しく階層移動の検出・削除の段階を追加する．よって，変更前後の AST の差分を計算するアルゴリズムなど GumTree の既存のアルゴリズムはそのまま用いる．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．

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），移動した部分木の根ノードが 4.2 節で定義したノードの種類に含まれる場合（アルゴリズム 1，6-7 行目），ボトムアップとトップダウンによる 2 ステップによる判定を実施する．2 つのステップを両方ともクリアした場合に階層移動と判定され，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）．

ボトムアップでは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種類のノードを指す．例えば，図 6 の場合，移動前の部分木における根ノード（MethodInvocation）から全体の木における根ノードに向かって遡ると，異種類のノードである ReturnStatement に到達する．移動後の部分木における根ノード（MethodInvocation）から全体の木における根ノードに向かって遡ると，同じ種類のノードである InfixExpression を通過し，異種類のノードである 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は Statement よりも大きなノード単位で移動が生じたことになるため，階層移動とは見なさない．

アルゴリズム 2: トップダウンによる経路判定アルゴリズム

Input: 先祖ノードから移動前後の部分木までの経路 ($route_1$, $route_2$)，GumTree が出力した変更前後の AST 間のマッピング集合 (M)

Output: 判定結果（true: 階層移動である，false: 階層移動ではない）

```
1  for node1 in route1 do
2      if node1 is not mapped then
3          continue;
4      node2 ← getMappedNode(node1);
5      if node2 ∉ route2 then
6          continue;
7      nextNode1 ← getNextNode(node1);
8      nextNode2 ← getNextNode(node2);
9      if childProperty(nextNode1) != childProperty(nextNode2) then
10         return false;
11     if childProperty(nextNode1) consists of multiple nodes then
12         if nextNode1 and nextNode2 have gaps in sibling nodes then
13             return false;
14 return true;
```

図 13　横移動が検出された変更前後のソースコード例

次に，トップダウンによる経路判定を実施する（アルゴリズム 1，10 行目）．経路判定では，ボトムアップによって到達した先祖ノードから移動前後の部分木に至るまでの経路を比較し，横移動を検出する．4.1 節に定義した通り，横移動を伴う移動は提案手法における検出対象外である．移動前の経路における先祖ノードから順に確認し，全てのノードにおいて横移動を検出しなかった場合に階層移動であると判定する．アルゴリズム 2 に横移動判定アルゴリズムの概要を示す．

移動前の経路を $route_1$，移動後の経路を $route_2$ とし，$route_1$ 上の対象のノードを $node_1$ とする．まず，$node_1$ と対応関係にあるノードが変更後の AST に存在するかどうかを判定する．$node_1$ と対応関係にあるノードが変更後の AST に存在しない場合，次のノードの判定に進む（アルゴリズム 2，2-3 行目）．次に，$node_1$ と対応関係にあるノードが変更後の AST に存在する場合に，対応するノードが $route_2$ に存在するかを判定する．対応するノードが $route_2$ に存在しない場合，次のノードの判定に進む（アルゴリズム 2，5-6 行目）．これら 2 つの判定では $node_1$ が変更前の経路から消えたかどうかを確認している．変更前後の経路においてノードの挿入・削除が発生した場合，単に上下移動が生じているだけであり，横移動の心配はない．移動を伴う経路上へのノードの挿入・削除も同様である．$node_1$ と対応関係を持つノード $node_2$ が変更後の AST に存在し，かつ $node_2$ が $route_2$ 上に存在する場合，次の判定に進む．

次に，経路上の次のノード $nextNode_1$ と $nextNode_2$ がそれぞれ同じ子プロパティを持つかどうかを判定する．$nextNode_1$ と $nextNode_2$ がどちらも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（アルゴリズム 2，9-10 行目）．例えば，図 13 に示す変更前後のソースコードでは `nodes[i]` が横移動しており明らかに移動を確認できるため階層移動とは見做さない．図 14 は図 13 における変更前後のソースコードをそれぞれ AST で表している．共通の先祖ノード ReturnStatement から移動前後のそれぞれの部分木までの経路上のノードだけを見ると，追加された CastExpression を除いて一致しているように見える．実際には MethodInvocation を境に異なる経路に枝分かれしており，横移動と判定する必要がある．よって，子プロパティの情報を基に経路の枝分かれがないかを判定する必要がある．図 15 は，今回経路の枝分かれとなっている MethodInvocation の子ノードに子プロパティが付与されている．例えば，変更前の AST において経路上のノード ArrayAccess には expression が付与されている一方，変更後の AST では経路上のノード CastExpression に arguments が付与されている．このように，子プロパティが異なる場合同じノード内で経路が分岐していることになるため，横移動と判定する．このような子プロパティは 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

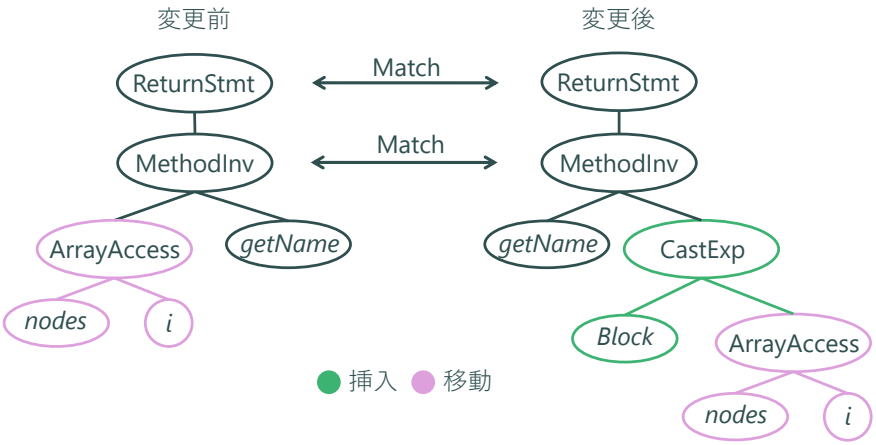

図 14　横移動が検出されたソースコードにおける変更前後の AST

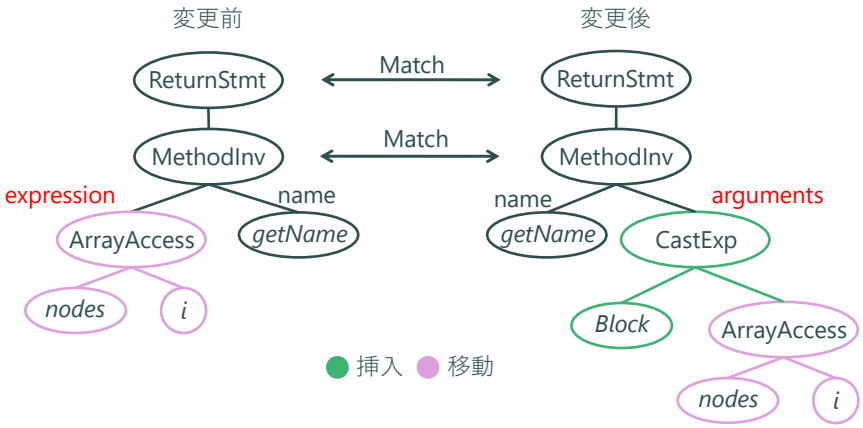

図 15　子プロパティを追加した変更前後の AST

を付与した．

一方，$nextNode_1$ と $nextNode_2$ が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ことになる．図 16 はノード MethodInvocation が複数の引数を持っており，引数のノード間で横移動が発生した例である．このような同じ子プロパティにおける枝分かれを判定するために，それぞれの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）を使用したずれの判定を実施する（アルゴリズム 2，11-13 行目）．図 17 は同じ子プロパティを持つトークン列において，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．対応関係のあるノード列の LCS を基準に並べ替えると図 18 の様になる．図 18 からは LCS として計算されたノードを基準に経路上のノードの位置が変わっていることが分かる．経路上のノードの位置にずれが生じる場合，経路の枝分かれが生じており横移動が生じていると判定する．一方，図 19 はずれが生じていない例であり，横移動とは判定されない．

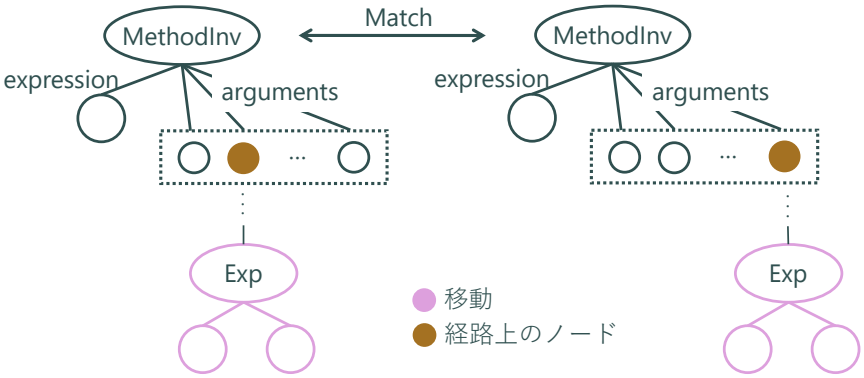

図 16　同じ子プロパティを持つノード間で枝分かれが生じている例

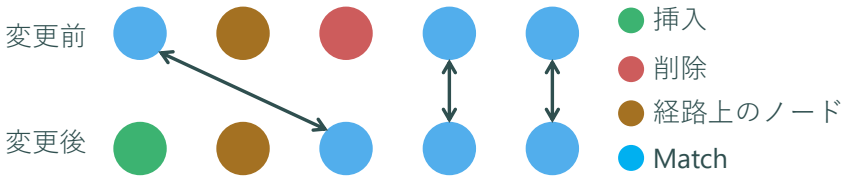

図 17　同じ子プロパティを持つノード列に対する対応ノードの LCS 結果

図 18　LCS の結果を基準に並べ替えたノード列

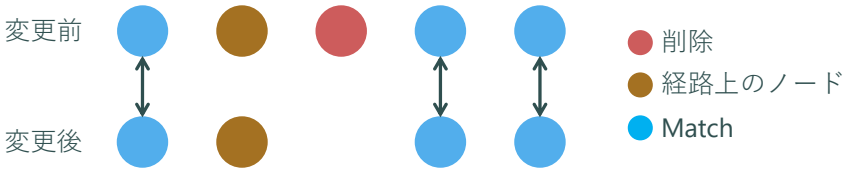

図 19　同じ子プロパティを持つノード列でずれが発生しない例

5　評価

本節では評価で使用したデータセットと評価手法について述べる．提案手法の評価を実施するために次の 4 つの Research Question 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のか？
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 上で理解できるのか？

RQ4 従来の GumTree と比較し，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のか？

RQ1 では本研究で定義した階層移動が実際のソースコード変更においてどの程度存在するかを確認する．これにより，提案手法を実際にソースコード差分検出に用いた際，検出結果が変化する割合を評価する．RQ2 では実際のソースコード変更において検出した階層移動における構造的な変更パターンを分類し，その知見を共有することを目的としている．RQ3 では提案手法と既存手法がそれぞれ出力した差分を開発者が正確に理解できるかを評価する．これにより，提案手法によって階層移動を削除した差分が差分理解に負の影響を与えないこと，さらに従来の GumTree が出力した移動情報の原因を開発者が正しく理解できるのかを検証する．RQ4 では，開発者にとって提案手法による差分出力と既存手法による差分出力はどちらの方が理解しやすいかを評価する．

5.1 データセット

本評価では既存手法である GumTree[7] 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[15] 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[7] 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 評価手法

本研究では 3 種類の評価手法 (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．

1 つ目はコンピュータを使用した自動評価である．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，以下の項目を測定した．

- 階層移動を検出したファイルの組数とその割合
- 階層移動の検出数と総移動数との割合
- 検出された階層構造のノードの所属グループ毎の数と割合（4.2 節で定義したノードの種類別分類）

2 つ目は著者によるマニュアル評価である．300 組（内 Defects4J，GhJava それぞれ 150 組ずつを含む）の変更を対象に検出した階層移動における変更パターンをより詳細に分類する．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．

3 つ目は被験者参加型の被験者評価である．評価する項目は以下の 3 つである．

- 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．
- 開発者はどの程度 AST に馴染みがあるのか．
- 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．

実験では，マニュアル評価で分類したパターンにつき 1 つの変更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，被験者に対して以下の 2 つの質問を実施した．

- 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．
- 与えられた差分はどちらの手法の方が理解しやすいか．5 段階での評価（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）

参加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以下の 3 つの対策を施した．

- 提案手法が有利にならないように，どちらの差分が提案手法かは被験者には伝えない．
- 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，参加者を 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る．
- 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，参加者によって差分の見せる順番をランダムに入れ替える．

最後に，AST の理解度を調査するために被験者に対して Java の簡単なソースコード（図 3 の変更後のソースコード）を見せ，AST を記述するよう依頼した．その後，著者が AST を確認し，正しく記載できているかを判定した．

6　評価結果

6.1　自動評価結果（RQ1）

2045 組の変更前後のファイルに対して提案手法で階層移動の削除を行ったところ，348 組（17.0%）から階層移動が検出・削除された．また，全ての組における移動総数 3172 個の内 540 個（17.0%）が階層移動と判定され，削除されることになった．この 540 個の移動を 4.2 節で定義した 5 種類に分類すると表 2 のような結果になった．最も多いのは Expression グループ内での階層移動であり，if 文における else 節，Name グループ，Type グループと続く．Pattern に関しては構文的に階層移動が発生してもおかしくないが，用意したデータセットからは検出されなかった．これはそもそも 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．

RQ1 への回答 変更前後のファイルの組における 17% に階層移動が含まれ，これは総移動数の 17% を占める．また，Expression グループ内での移動が最も多かった．

表 2　自動評価による階層移動の所属グループ別分類結果

グループ名	移動数（個）	割合（%）
Expression	446	82.6
Name	23	4.3
Pattern	0	0
Type	12	2.2
if 文の else 節	59	10.9

表 3　マニュアル評価による階層移動のパターン分類結果

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

6.2　マニュアル評価結果（RQ2）

マニュアル評価では計 300 組の変更前後のファイルの組から検出された 106 個の階層移動を目視で分類した．分類の結果，12 種類の変更パターンに分類できた．表 3 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．中には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．

RQ2 への回答 300 個の変更前後のファイルの組から検出された階層移動を手作業で分類したところ，12 種類の変更パターンに分類された．この内，四則演算や論理演算における変更が最も多く，メソッドの引数へ（から）の変更が 2 番目に多かった．

6.3　被験者評価結果（RQ3，RQ4）

6.2 節で分類した 12 種類のパターンにおいて，変更前後のファイルの組を 1 つずつランダムに選び，提案手法と既存手法両方で出力した Diff View をそれぞれ被験者に提示した．その結果，既存手法が出力した 6 種類の階層移動の差分において，参加者の過半数が移動の原因を正しく解凍することができなかった．一方，提案手法が検出した階層移動を除いた Diff View では全てのパターンで差分について正確に説明できた．表 4 はどの程度の被験者が既存手法が出力した階層移動の原因を正確に理解したかを表している．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

表 4　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（人）	理解していない人数（人）
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

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大きな悪影響を及ぼすと言える．

また，AST の理解度を測るために簡単なソースコードの AST を記述してもらったところ，10 人中 5 人の被験者がすぐには AST を記述することができなかった．コンピュータサイエンスを専攻する学生（内学部 4 年が 2 人，修士の学生が 8 人）の半数が AST に馴染みがないということが分かった．この事実から，一般的な開発者の多くはソースコードから AST を想像することが難しく，ソースコード上で認知できる差分情報から逸脱した AST 特有の差分は除外すべきであるということが言える．

RQ3 への回答 RQ2 で分類した 12 種類の内 6 種類において移動の原因を理解できない人が半数を超えた．さらに，半数の被験者が AST に馴染みが薄いということが分かった．これらの結果から，本研究で定義した階層移動は Diff View で確認するとその移動原因が分かりずらく，出力を避けるべきであるということが言える．

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果，表 5 に示す結果になった．アンケートの結果，パターン Exp-7 を除く 11 種類のパターンにおいて，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，参加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，全ての参加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という結果になった，三項演算へ（から）の

表 5　提案手法と既存手法のアンケート結果

		参加者											
		A	B	C	D	E	F	G	H	I	J	平均値	中央値
パターン	Exp-1	1	1	1	1	1	1	1	1	1	1	1.00	1.0
	Exp-2	1	1	3	3	2	3	2	4	1	3	2.30	2.5
	Exp-3	1	1	1	1	1	1	1	1	1	1	1.00	1.0
	Exp-4	1	1	1	1	2	2	1	1	1	2	1.30	1.0
	Exp-5	1	1	3	1	1	5	2	1	2	1	1.80	1.0
	Exp-6	1	1	1	1	1	2	2	1	1	1	1.20	1.0
	Exp-7	1	1	4	5	5	5	2	2	5	4	3.40	4.0
	Name-1	1	1	1	1	1	1	1	1	1	1	1.00	1.0
	Type-1	1	2	1	1	1	1	2	1	1	1	1.20	1.0
	Type-2	1	3	1	1	2	2	3	1	1	2	1.70	1.5
	If-1	1	1	1	5	2	1	1	5	1	4	2.20	1.0
	If-2	1	1	1	1	2	1	1	1	1	4	1.40	1.0
	平均値	1.00	1.25	1.58	1.83	1.75	2.08	1.58	1.67	1.42	2.08		
	中央値	1.0	1.0	1.0	1.0	1.5	1.5	1.5	1.0	1.0	1.5		

変更前

```
return
  TypeHandler.createValue(res, value);
```

変更後

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

図 20　既存手法に軍配が上がった三項演算における階層移動差分例（既存手法）

変更前

```
return
  TypeHandler.createValue(res, value);
```

変更後

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

図 21　既存手法に軍配が上がった三項演算における階層移動差分例（提案手法）

移動（Exp-7）における差分例を図 20，21 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により階層移動が生じたため，既存手法による Diff View では移動が確認できる．一方，提案手法ではこのコード片における階層移動を検出・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されたい．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，参加者にとっては Diff View に表示した方が理解しやすかったようだ．参加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既存手法による差分の方が理解しやすいという声が多くの参加者から上がった．

RQ4 への回答 RQ2 で分類した 12 種類の階層移動の内，11 種類の差分において提案手法の方が開発者にとって理解しやすいという結果になった．一方，残りの 1 種類である三項演算の要素へ（から）の変更については，既存手法の方が理解しやすいという結果を得た．

7　妥当性への脅威

本研究では階層移動を著者らの基準で予め定義し，その上で検出を行った．よって，開発者にとって不必要な階層移動を全て定義・検出できているわけではない．しかし，AST において発生しうる階層移動を可能な限り議論し，多くの変更前後のソースコードへの適用・調査を実施した．

また，提案手法は GumTree の性能に依存している．提案手法では GumTree が出力した編集スクリプトにおける移動情報を元に階層移動を検出する．よって，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．しかし，GumTree における差分検出アルゴリズム自体の性能向上に関しては本研究の対象外であり，今後別の研究として注力すべき分野である．

評価で実施した手作業によるパターン分類は我々の基準で行った．分類に明確な基準は存在しないため，必ずしも適切であるとは言えない．また，今回は 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．

被験者評価では，時間の制約上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．しかし，出来るだけ多くのパターンにおいて開発者の認識を調査するために 12 種類全てのパターンを対象に実験を実施した．

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．

8 関連研究

これまで多くのソースコードの差分検出手法が提案されてきた．ソースコードの差分検出手法は大きく分けて，行やトークンなどのテキストベースの差分検出手法と AST を使用した構造上の差分検出手法の 2 つに分類される [16].

行単位の差分検出ツールとして Unix の diff コマンドや Git の diff サブコマンドが有名であり，2 つの異なるファイル間の差分を行の挿入と削除で表現する．これらの内部では Myers[10] や Histogram などのアルゴリズムが使用されており，Git の diff サブコマンドでは複数の種類からアルゴリズムを選択可能である [11]．diff コマンドの他にも，挿入と削除だけでなく，行を追跡することで移動を検出する手法も提案されている [4, 17, 18]．さらに，行よりも細粒度であるプログラムのトークンに着目したツールも提案されている．例えば cregit[19] はトークンレベルでソースコードの差分を検出することでプログラムの実装者の追跡精度を向上させた．さらに，FinerGit[20] はトークンレベルで差分を検出することでメソッドの追跡精度を向上させた.

AST を使用した構造上の差分検出手法も複数提案されており, ChangeDistiller[8] や Diff/TS[9], GumTree[6, 7] などがある．ChangeDistiller[8] は既存の階層構造の差分検出アルゴリズム [21] を基に提案された手法であるが，AST の葉ノードが行レベルであり差分の粒度が粗い．Diff/TS[9] は既存の木構造の編集距離検出アルゴリズム [22] を基に提案されており，AST をそのまま用いて差分を検出可能である．しかし，これらの AST を用いた差分検出手法において，移動や更新を含んだ編集スクリプトを検出するのは NP 困難であることが知られている [23]．本研究でベンチマークとした GumTree[6, 7] は，Cobena らの手法 [24] の手法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 [14] 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 [25] 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[26] 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 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[27] 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[28] 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手法は本研究が初である.

さらに，テキストベースの差分検出手法と AST を用いた差分検出手法を組み合わせたハイブリッドな手法も提案されている．Matsumoto ら [29] は行単位の差分と AST の差分を組み合わせることで編集スクリプトを短くし，より理解しやすい差分を出力する手法を提案した．Yang ら [30] らは行単位の差分から得られるハンクを利用することで AST 間の差分計算における実行時間を短縮した.

9　後書き

本研究では，変更前後の AST における階層移動に着目し，GumTree が出力する差分情報と開発者がソースコードから読み取れる差分情報における認識の乖離を改善した．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 種類全ての変更パターンにおいて被験者実験を実施したところ，11 種類が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

まずは弊研究室の教授である肥後先生にお礼を申し上げる．肥後先生からは物事の進め方を学んだ．普段から「肥後先生なら何を思うか」を常に想像しながら研究に取り組むことで，論理的に道筋を立てて適切に物事を進める能力が向上した．また，行き詰まった際に助言を頂いたことで進むべき方向を再確認できた．3 年間に渡りご指導いただき，ありがとうございました．

次に指導教官である松下先生にお礼を申し上げたい．松下先生には本研究を遂行するにあたり，研究テーマの発案から手法の提案，評価方法など一通りご指導いただいた．特に，毎週個別にミーティングを組んでいただき，研究の進捗や直面した課題の相談に乗っていただいた．常に暖かくサポートいただいたことで自分が本当にやりたい研究テーマに出会うことができ，伸び伸び取り組むことができた．学部 4 年の時から 3 年間に渡り直接指導いただきき，ありがとうございました．

And I want to thank you to professor Raula. He joined our laboratory as a professor in the summer of 2024. Although it was short-term, I was greatly helped by his cheerful personality. While there were only a few opportunities for him to directly advise me on my research, the discussions during our group seminars helped deepen my understanding. Thank you very much for this past half year.

次に招へい教員である神田先生にお礼を申し上げる．神田先生は弊研究室在籍中の 2 年間と他大学へ転籍後の 1 年間に渡りご指導いただいた．先生の中で最も学生との距離が近く，気軽に会話できた．中間報告会などで暖かい助言をいただいたことで安心して研究に取り組むことができた．3 年間に渡りご指導いただき，ありがとうございました．

次に研究室の事務員である軽部さんにお礼を申し上げる．軽部さんの明るく暖かい人柄は研究生活を送る上で非常に支えになった．研究室ではいつも軽部さんの明るい声が聞こえ，その声を聞くだけで元気になれた．また，研究室のイベントがあるたびに率先して企画していただき沢山の明るい思い出を作ることができた．3 年間に渡りご支援いただき，ありがとうございました．

次に，研究生活を共にした研究室のメンバーにお礼を申し上げる．非常に優秀な仲間に恵まれ，快適な研究室生活を送ることができた．研究室では日常会話から技術的な内容まで様々なことを話し，沢山のことを学んだ．特にソフトウェア工学を専攻していることもあり，プログラミングや開発の話題が多くとても楽しかった．大学院生活の苦楽を共にした仲間であるため今後も出来れば交流を続けていきたいと思う．一旦の節目として感謝を申し上げたい．ありがとうございました．

次に，研究生活で関わっていただいた全ての方々にお礼を申し上げる．研究室外の発表や交流の場でご意見，ご指導を頂いた皆さん，ありがとうございました．

最後に，大学院まで通わせてくれた家族にお礼を言いたい．大学院に進学することに肯定的で，心良く送り出してくれた．金銭面でも精神面でも十分すぎる支援を受け，何一つ不自由なく研究生活を送ることができた．家族の皆ありがとう．

参考文献

[1] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the optimal order of reading source code changes for review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 329–340, 2017.

[2] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[3] J.I. Maletic and M.L. Collard. Supporting source code difference analysis. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 210–219, 2004.

[4] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *2009 IEEE 31st International Conference on Software Engineering*, pages 595–598, 2009.

[5] 大森隆行 and 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**, 49(7):2349–2359–2359, 7 2008.

[6] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Jean-Remy Falleri and Matias Martinez. Fine-grained, accurate and scalable source differencing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[8] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[9] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th Working Conference on Reverse Engineering*, pages 279–288, 2008.

[10] E.W. Myers. Ano(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[11] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. How different are different diff algorithms in git? use–histogram for code changes. *Empirical Software Engineering*, 25:790–823, 2020.

[12] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on*

Foundations of Software Engineering, ESEC/FSE 2017, page 140–150, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. A look into programmers' heads. *IEEE Transactions on Software Engineering*, 46(4):442–462, 2020.

[14] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. Generating simpler ast edit scripts by considering copy-and-paste. In *The 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE2017)*, pages 532–542, 10 2017.

[15] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 298–309, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 58–64, New York, NY, USA, 2006. Association for Computing Machinery.

[17] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *2013 IEEE International Conference on Software Maintenance*, pages 230–239, 2013.

[18] Steven P. Reiss. Tracking source locations. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 11–20, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Daniel M. German, Bram Adams, and Kate Stewart. cregit: Token-level blame information in git version control repositories. *Empirical Softw. Engg.*, 24(4):2725–2763, August 2019.

[20] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. On tracking java methods with git mechanisms. *Journal of Systems and Software*, 165:110571, 2020.

[21] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 493–504, New York, NY, USA, 1996. Association for Computing Machinery.

[22] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262, 12 1989.

[23] Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1):217–239, 2005.

[24] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Proceedings 18th International Conference on Data Engineering*, pages 41–52, 2002.

[25] Akira Fujimoto, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. Staged tree matching for detecting code move across files. In *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, pages 396–400, 7 2020.

[26] Georg Dotzler and Michael Philippsen. Move-optimized source code tree differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 660–671, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Michael John Decker, Michael L Collard, L Gwenn Volkert, and Jonathan I Maletic. srcdiff: A syntactic differencing approach to improve the understandability of deltas. *Journal of Software: Evolution and Process*, 32(4):e2226, 2020.

[28] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. Generating accurate and compact edit scripts using tree differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 264–274, 2018.

[29] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Beyond gumtree: A hybrid approach to generate edit scripts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 550–554, 2019.

[30] Chunhua Yang and E. James Whitehead. Pruning the ast with hunks to speed up tree differencing. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 15–25, 2019.

付録 A　JDT の AST ノード分類結果

JDT における AST ノードの分類結果を，クラスの継承関係を基に以下に示す．クラス名の後ろに (A) が付いているノードは JDT のソースコードで抽象クラスとして実装されており，実際には AST のノードとしては出現しない．その代わりに，そのクラスを継承する子クラスをまとめるグループ名となる．また，クラスの継承関係はインデントを用いた階層構造で表している．例えば Expression クラスは抽象クラスであり，Annotation から VariableDeclarationExpression までの全てのノードを子クラスとして持つ．つまり，Annotation から VariableDeclarationExpression までの全ノードが Expression グループに属することになる．次に，クラス名の後ろに角括弧`[]`が存在する場合，その AST ノードが自分が属するグループのノードを子ノードとして持ちうるということを示す．例えば，ArrayAccess の後ろには`[Expression]`と記載があるが，これはノード ArrayAccess が自分が属する Expression グループのノードを子ノードとして持ちうることを示す．

- AnonymousClassDeclaration
- BodyDeclaration(A)
  - AbstractTypeDeclaration(A)
    - AnnotationTypeDeclaration [BodyDeclaration]
    - EnumDeclaration
    - ImplicitTypeDeclaration
    - RecordDeclaration
    - TypeDeclaration
  - AnnotationTypeMemberDeclaration [BodyDeclaration]
  - EnumConstantDeclaration
  - FieldDeclaration
  - Initializer
  - MethodDeclaration
- CatchClause
- CompilationUnit
- Dimension
- Expression(A)
  - Annotation(A)
    - MarkerAnnotation
    - NormalAnnotation
    - SingleMemberAnnotation [Expression]
  - ArrayAccess [Expression]
  - ArrayCreation [Expression]
  - ArrayInitializer [Expression]

- Assignment [Expression]
- BooleanLiteral
- CaseDefaultExpression [Expression]
- CastExpression [Expression]
- CharacterLiteral
- ClassInstanceCreation [Expression]
- ConditionalExpression [Expression]
- FieldAccess [Expression]
- InfixExpression [Expression]
- InstanceofExpression [Expression]
- LambdaExpression [Expression]
- MethodInvocation [Expression]
- MethodReference [Expression]
- Name(A)
  - ModuleQualifiedName
  - QualifiedName [Name]
  - SimpleName
- NullLiteral
- NumberLiteral
- ParenthesizedExpression [Expression]
- Pattern(A)
  - EitherOrMultiPattern [Pattern]
  - GuardedPattern
  - NullPattern
  - RecordPattern [Pattern]
  - TypePattern
- PatternInstanceofExpression [Expression]
- PostfixExpression [Expression]
- PrefixExpression [Expression]
- StringLiteral
- SuperFieldAccess [Expression]
- SuperMethodInvocation [Expression]
- SwitchExpression [Expression]
- TextBlock
- ThisExpression [Expression]
- TypeLiteral
- VariableDeclarationExpression [Expression]

- ImportDeclaration
- MemberValuePair
- MethodRefParameter
- Modifier
- ModuleDeclaration
- ModuleDirective(A)
  - ModulePackageAccess(A)
    - ExportsDirective
    - OpensDirective
  - ProvidesDirective
  - RequiresDirective
  - UsesDirective
- ModuleModifier
- PackageDeclaration
- Statement(A)
  - AssertStatement
  - Block [Statement]
  - BreakStatement
  - ConstructorInvocation
  - ContinueStatement
  - DoStatement [Statement]
  - EmptyStatement
  - EnhancedForStatement [Statement]
  - ExpressionStatement
  - ForStatement [Statement]
  - IfStatement [Statement]
  - LabeledStatement [Statement]
  - ReturnStatement
  - SuperConstructorInvocation
  - SwitchCase
  - SwitchStatement [Statement]
  - SynchronizedStatement（Block）
  - ThrowStatement
  - TryStatement（Block）
  - TypeDeclarationStatement
  - VariableDeclarationStatement
  - WhileStatement [Statement]

    - YieldStatement
- Type(A)
    - AnnotatableType(A)
        - NameQualifiedType
        - PrimitiveType
        - QualifiedType [Type]
        - SimpleType
        - WildcardType [Type]
    - ArrayType [Type]
    - IntersectionType [Type]
    - ParameterizedType [Type]
    - UnionType [Type]
- TypeParameter
- VariableDeclaration
    - SingleVariableDeclaration
    - VariableDeclarationFragment

付録 B　パターン別の差分代表例

変更前

```
raw == String.class || raw == Object.class
```

変更後

```
raw == String.class || raw == Object.class || raw == CharSequence.class
```

図 22　Exp-1: 既存手法による差分出力

変更前

```
raw == String.class || raw == Object.class
```

変更後

```
raw == String.class || raw == Object.class || raw == CharSequence.class
```

図 23　Exp-1: 提案手法による差分出力

変更前

```
"" + eachArg.charAt(i)
```

変更後

```
String.valueOf(eachArg.charAt(i))
```

図 24　Exp-2: 既存手法による差分出力

変更前

```
"" + eachArg.charAt(i)
```

変更後

```
String.valueOf(eachArg.charAt(i))
```

図 25　Exp-2: 提案手法による差分出力

変更前

```
DEFAULT.withIgnoreEmptyLines(false)
```

変更後

```
DEFAULT.withIgnoreEmptyLines(false).withAllowMissingColumnNames(true)
```

図 26　Exp-3: 既存手法による差分出力

変更前

```
DEFAULT.withIgnoreEmptyLines(false)
```

変更後

```
DEFAULT.withIgnoreEmptyLines(false).withAllowMissingColumnNames(true)
```

図 27　Exp-3: 提案手法による差分出力

変更前

```
(double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()
```

変更後

```
getSampleSize() * (getNumberOfSuccesses() / (double) getPopulationSize())
```

図 28　Exp-4: 既存手法による差分出力

変更前

```
(double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()
```

変更後

```
getSampleSize() * (getNumberOfSuccesses() / (double) getPopulationSize())
```

図 29　Exp-4: 提案手法による差分出力

変更前

```
tag.isSelfClosing()
```

変更後

```
!tag.isEmpty()
```

図 30　Exp-5: 既存手法による差分出力

変更前

```
tag.isSelfClosing()
```

変更後

```
!tag.isEmpty()
```

図 31　Exp-5: 提案手法による差分出力

変更前

```
(Option) options.getOption(arg)
```

変更後

```
options.getOption(ch)
```

図 32　Exp-6: 既存手法による差分出力

変更前

```
(Option) options.getOption(arg)
```

変更後

```
options.getOption(ch)
```

図 33　Exp-6: 提案手法による差分出力

変更前

```
return
  TypeHandler.createValue(res, value);
```

変更後

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

図 34　Exp-7: 既存手法による差分出力

変更前

```
return
  TypeHandler.createValue(res, value);
```

変更後

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

図 35　Exp-7: 提案手法による差分出力

変更前

```
import com.ning.billing.entitlement.api.user.ISubscription;
```

変更後

```
import static com.ning.billing.entitlement.api.user.Isubscription.SubscriptionState;
```

図 36　Name-1: 既存手法による差分出力

変更前

```
import com.ning.billing.entitlement.api.user.ISubscription;
```

変更後

```
import static com.ning.billing.entitlement.api.user.Isubscription.SubscriptionState;
```

図 37　Name-1: 提案手法による差分出力

変更前

```
private final Constraint constraint;
```

変更後

```
private final Constraint[] constraints;
```

図 38　Type-1: 既存手法による差分出力

変更前

```
private final Constraint constraint;
```

変更後

```
private final Constraint[] constraints;
```

図 39　Type-1: 提案手法による差分出力

変更前

```
private final HashMap<String, String> namespaces = new HashMap<>();
```

変更後

```
private final Stack<HashMap<String, String>> namespacesStack = new Stack<>();
```

図 40　Type-2: 既存手法による差分出力

変更前

```
private final HashMap<String, String> namespaces = new HashMap<>();
```

変更後

```
private final Stack<HashMap<String, String>> namespacesStack = new Stack<>();
```

図 41　Type-2: 提案手法による差分出力

変更前

```
if (options.hasOption(arg.substring(0, 2)))
{
    ...
}
else
{
    ...
}
```

変更後

```
if (opt.indexOf('=') != -1 && ...)
{
    ...
}
else if (options.hasOption(arg.substring(0, 2)))
{
    ...
}
else
{
    ...
}
```

図 42　IfStatement-1: 既存手法による差分出力

変更前

```
if (options.hasOption(arg.substring(0, 2)))
{
    ...
}
else
{
    ...
}
```

変更後

```
if (opt.indexOf('=') != -1 && ...)
{
    ...
}
else if (options.hasOption(arg.substring(0, 2)))
{
   ...
}
else
{
    ...
}
```

図 43　IfStatement-1: 提案手法による差分出力

変更前

```
if (token == JsonToken.FIELD_NAME) {
    ...
} else if (token == JsonToken.START_OBJECT) {
    ...
} else if (token == JsonToken.VALUE_TRUE ...) {
    ...
} else {
    ...
}
```

変更後

```
if (token == JsonToken.FIELD_NAME) {
    ...
} else if (token == JsonToken.START_OBJECT) {
    ...
} if (token == JsonToken.START_ARRAY) {
    ...
} else if (token == JsonToken.VALUE_TRUE ...) {
    ...
} else {
    ...
}
```

図 44　IfStatement-2: 既存手法による差分出力

変更前

```
if (token == JsonToken.FIELD_NAME) {
    ...
} else if (token == JsonToken.START_OBJECT) {
    ...
} else if (token == JsonToken.VALUE_TRUE ...) {
    ...
} else {
    ...
}
```

変更後

```
if (token == JsonToken.FIELD_NAME) {
    ...
} else if (token == JsonToken.START_OBJECT) {
    ...
} if (token == JsonToken.START_ARRAY) {
    ...
} else if (token == JsonToken.VALUE_TRUE ...) {
    ...
} else {
    ...
}
```

図 45　IfStatement-2: 提案手法による差分出力