特別研究報告

題目

コードクローンに対するリファクタリング可能性判定手法の改善

指導教員
井上 克郎 教授

報告者
桑谷 実

令和 2 年 2 月 10 日

大阪大学 基礎工学部 情報科学科

平成 31 年度 特別研究報告

コードクローンに対するリファクタリング可能性判定手法の改善

桑谷 実

内容梗概

コードクローンとは,「ソースコード中に存在する互いに一致または類似した部分を持つコード片」のことである．コードクローンは，コピーアンドペースト等の既存のコード片の再利用により生じる．また，コードクローンはソフトウェア保守を困難にさせる要因の 1 つとされている．例えば，修正が必要な箇所がコードクローンを持つコード片で発見された場合，コードクローンの関係にあるコード片にも修正が必要になる可能性が高い．そのため，開発者によって意図的に残されている場合を除き，コードクローンは可能な限り取り除くことが望まれる．コードクローンを取り除く手法の 1 つとしてリファクタリングがある．

リファクタリングとは,「可読性や保守性の向上を目的としてソフトウェア外部の振る舞いを保ったまま内部構造を改善すること」である．コードクローンに対してリファクタリングを行う場合，コードクローンをメソッドやクラスに集約する作業によって行われる．リファクタリングによりコードクローン内の修正箇所が集約されるため，保守コストを抑えることが可能である．ただし，コードクローンのリファクタリングにかかるコストは対象となるコードクローン同士の相違点や位置関係によって異なり，あらゆるコードクローンに対して同等というわけではない．

リファクタリングの支援を行うツールは多数存在しており，その中の 1 つに JDeodorant というものがある．JDeodorant は，Java プロジェクトに対してリファクタリング支援を行う Eclipse プラグインである．プロジェクト内に存在する「コードクローンの関係にあるコード片の組」であるクローンペアに対して，リファクタリングを提案することが JDeodorant の主な機能である．また，JDeodorant には，クローンペアのリファクタリング可能性を判定する機能も備わっている．クローンペアのリファクタリング可能性は「refactorable(リファクタリング可能)」もしくは「non-refactorable(リファクタリング困難)」で判定される．

本研究では，リファクタリング困難だと判定されているコードクローンのうち，実際にはリファクタリングが可能なものに注目し，リファクタリング可能性の誤判定が起きる原因を解消するコード変換を既存の手法に追加することでリファクタリング可能性判定の改善を行う．

評価実験では誤判定の起きたコード片をもとに作成したデータセットに対して、リファクタリング可能性判定を行い，データセット内のクローンペアのうちリファクタリング可能の

判定を受けたクローンペアの割合は 0.28 から 0.51 に増加し，誤判定の一部に対応できることを確認した.

主な用語

コードクローン
リファクタリング
コード変換
リファクタリング可能性

目 次

1　まえがき

コードクローンとは,「ソースコード中に存在する互いに一致または類似した部分を持つコード片」のことである.[1] コードクローンの主な発生原因はコピーアンドペースト等の既存のコード片の再利用である．もし，不具合の解消や機能の改善のためにコードクローンをもつコード片の修正を行う際，コードクローンの一部を見逃して修正が行われなかった場合，開発者の意図しない動作が起きてしまう可能性がある．もしくは，修正が必要な箇所が含まれたコード片を再利用した場合，修正を必要とする箇所を含んだままのコード片がソースコードに散在してしまうことになる．このようにコードクローンの生成によりソースコードの修正に大きなコストが必要になり，コードクローンはソフトウェア保守を困難にさせる要因の 1 つだともされている.[1] そのため，開発者の意図していないコードクローンはソースコード中から取り除くことが必要になりうる．コードクローンを取り除く手法の 1 つにリファクタリングが存在する.

リファクタリングとは「ソフトウェア外部の振る舞いを保ったまま，内部構造を改善すること」である。[2] コードクローンにおけるリファクタリングは，コードクローンをメソッドやクラスに集約することで実現する．具体的にはコードクローンと同等の機能をもつ新規のメソッドやクラスを作成し，コードクローン自身は呼び出し文に置換する．ただし，コードクローンにも完全に一致してるか違いを含んでいるか，コードクローンの場所が同じクラスやディレクトリ内にあるかそうでないかなどの条件の違いがあり，すべてのコードクローンに対して同等のコストでリファクタリングを行えるわけではない．そのため，コードクローンのリファクタリングの実行しやすさは対象となるコードクローンによって異なる.

「コードクローンの関係にあるコード片の組」であるクローンペアにおけるリファクタリングの実行しやすさをリファクタリング可能性として判定することができる JDeodorant[3] というツールが存在する．JDeodorant は Tsantalis らが開発したリファクタリング支援を行う Eclipse プラグインであり，クローン検出ツールの出力をもとにクローンペアのリファクタリングの提案などの機能を持つ．JDeodorant の機能の 1 つにクローンペアのリファクタリング可能性の判定機能がある．JDeodorant のリファクタリング可能性判定によってクローンペアは「refactorable(リファクタリング可能)」もしくは「non-refactorable(リファクタリング困難)」のどちらかだと判定される．(ただし，コードクローンが複数のメソッドを含む場合やその他不具合が生じた場合など判定を行わない場合もある.)

しかし，JDeodorant のリファクタリング可能性判定では，複雑でない作業でリファクタリングが行えるクローンペアの一部に対してリファクタリング困難と判定してしまう場合がある．すなわち，JDeodorant のリファクタリング可能性判定には，リファクタリング可能と判定されるべきクローンペアを誤ってリファクタリング困難と判定する可能性があるとい

う問題が存在する．

そこで，本研究では，実際にはリファクタリング可能の判定がされるべきであるのにリファクタリング困難だと判定されてしまったコードクローンに注目し，リファクタリング可能性の誤判定が起きる原因を解消するコード変換を既存の判定手法に追加することで，誤判定を改善したリファクタリング可能性判定手法を提案する．

本手法では，コード変換によりコードクローン内の誤判定を引き起こす箇所を取り除いた代替コードを生成し，代替コードに対してリファクタリング可能性判定を行うことで既存手法における誤判定を回避する．コード変換を行うために，変数の情報が必要になるので，対象となるソースコードを含むプロジェクト内のソースファイルに対して構文解析を行う．また，コードクローンの位置の情報も必要となるので，JDeodorant によるリファクタリング可能性の出力からコードクローンの開始行や終了行の情報を取り込む．これらの情報をもとに誤判定の原因となる箇所を特定し，コード変換を実行する．

評価実験では，誤判定の起きたコード片よりデータセットを生成し，既存手法と提案手法の両方のリファクタリング可能性判定の結果を比較する．既存手法において誤判定の起きるコード片を OSS より抽出し，そのコード片に対していくつかの変更を組み合わせて加えることで生成できるコード片の集合をデータセットとした．結果からどういった変更が提案手法の判定に影響するかを確認する．

2 章では，本研究の背景について述べる．3 章では，本研究で提案する手法について述べる．4 章では，本研究の評価実験について述べる．最後に，5 章でまとめと課題について述べる．

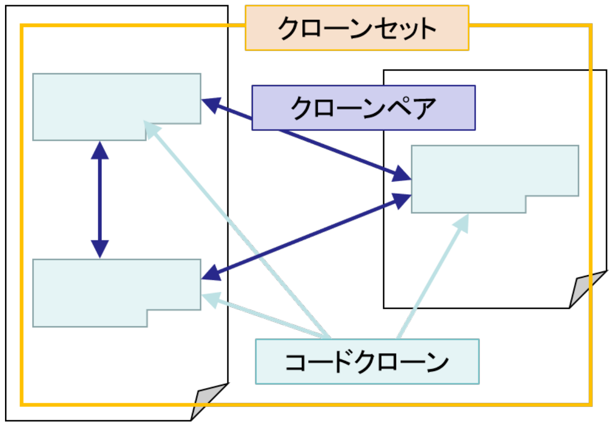

図 1: コードクローン

2　背景

本章では， 本研究の背景としてコードクローンの定義や，コードクローン検出ツールなどのリファクタリングの支援ツール，リファクタリング可能性判定における問題などについて述べる．

2.1　コードクローン

コードクローンとは,「ソースコード中に存在する互いに一致または類似した部分を持つコード片」のことである．コードクローンは，主にコピーアンドペーストなどの既存コードの再利用やツールによるコードの自動生成などによって生じる．また，コードクローンの関係にあるコード片の組をクローンペア，集合をクローンセットと呼ぶ.(図 1)

2.1.1　コードクローンの分類

コードクローンには普遍的定義が存在しない．Roy らはコードクローンの定義として，コードクローン間の違いに基づき，以下の 4 つのタイプに分類している.[4]

タイプ 1

空白, 改行, コメントなどの違いを除いて一致するコードクローン.

タイプ 2

タイプ 1 に加えて識別子, リテラル, 型の違いを除いて一致するコードクローン.

タイプ 3

タイプ 2 に加えて, 文の変更・挿入・削除などの違いを除いて一致するコードクローン.

タイプ 4

構文的な違いを含むが, 同様の処理を行うコードクローン.

2.1.2 コードクローン検出ツール

コードクローンを自動で検出する様々な技術がこれまでに提案されている. 既存の技術はコードクローンをどの単位で検出するかにより，以下の 5 つに分類することができる.[5]

行単位の検出

行単位の検出では，閾値以上連続して重複する行をコードクローンとして検出する.
Johnson や Ducasse らは，比較的単純な検出法を提案する.[6, 7, 8] この手法では空白やタブを除いたのち，行単位の比較を行う. 空白とタブの削除という単純な処理のみ行われるので，不特定多数の言語に対応可能である.
Roy らは，行単位の検出を行う NiCAD[9] を開発した.NiCAD ではオプションにより識別子の無視やソースコードの変形を行い，タイプ 3 までのコードクローンを検出が可能となっている.

字句単位の検出

字句単位の検出では，ソースコードを字句の列に変換した後，閾値以上連続する字句の部分列をコードクローンとして検出する. 字句の列の比較であるため，検出結果がコーディングスタイルに依存しない. また，検出のためにソースコードを中間表現に変換する必要がないため，高速なコードクローンの検出が可能である.
本研究では，コードクローン検出のために字句単位の検出ツールである CCFinderX[10] を活用した. CCFinderX は神谷らが開発した検出ツールで，特徴として関数名などのユーザー定義名を特殊文字に置き換えたのち検出処理を行う.

抽象構文木を用いた検出

抽象構文木を用いてコードクローンを検出では，ソースコードの構文解析により作成した抽象構文木上の部分木を比較し，同形の木をコードクローンとして検出する. プログラムの構造を無視したコードクローンを検出しない特徴を持つが，抽象構文木の構築が必要なため，比較的コストは高くなる.
抽象構文木を用いた検出ツールとして Baxter らは CloneDR[11] を開発した.CloneDR では部分木に対してハッシュ値を生成し，ハッシュ値が同一のもののみを比較する.
また，Jiang らは DECKARD[12] を開発した.DECKARD では部分木を特徴ベクトルに変換し，特徴ベクトルの類似度からコードクローンの検出を行う.

プログラム依存グラフを用いた検出

プログラム依存グラフを用いた検出では，ソースコードに意味解析を行い，ソースコードの要素の依存関係を抽出したプログラム依存グラフを構築し，同形の部分木をクローンとして検出する. この検出手法では意味的に同一なコードクローンを検出可能である. ただし，プログラム依存グラフは構築には高い計算コストが必要で，大規模ソフトウェアへの適用は非現実的である.

Komondoor らはソースコード中の文をプログラム依存グラフのノードとする検出手法を提案している.[13] この手法ではソースコードを種類に応じて分類し，同種の文の組にフォワードプログラムスライスとバックワードスライスを用いて作成したグラフの同一構造からコードクローンの検出を行う.

メトリクスなどの他の技術を用いた検出

上記に属さない検出技術も存在する. メトリクスを用いた検出では，プログラムのファイルやクラス，メソッドなどのモジュールに対してメトリクスを計測し，値の一致または近似の度合を検査し，モジュール単位のコードクローンを検出する.

Mayrand らは関数に対しる 21 種類のメトリクスの計測でコードクローンを検出する手法を提案する.[14] この手法ではユーザー定義名やコーディングスタイルなどに関するメトリクスを用いて，関数単位で 8 段階の類似度を定義している.

2.1.3 コードクローンのリファクタリング

リファクタリングとは「外部的な振る舞いを保ちつつ，内部構造の改善を行うこと」である. リファクタリングにはソフトウェアの保守性や可読性の向上を目的として，コードクローンを取り除く作業も含まれる. リファクタリングによりコードクローンを取り除く際は，図 2 に示すような作業を行う. コードクローンが行う処理を代替するメソッドやクラスを新規作成し，コードクローン自体はメソッドやクラスの呼び出し文に置換することでコードクローンを取り除くことができる.

2.2 リファクタリング支援ツール JDeodorant

Tsantalis らは Java 言語を対象としたリファクタリング支援ツールである JDeodorant を開発した. JDeodorant は Java 言語のソースコードを含む Eclipse プロジェクトとコードクローン検出ツールの出力を入力として，リファクタリングの支援を行う Eclipse プラグインである. コードクローン検出ツールは CCFinderX，DECKARD，CloneDR，Nicad に対応しており，本研究では CCFinderX を使用した. JDeodorant の機能の 1 つにリファクタリング可能性判定の機能が存在する. ネスト構造木とプログラム依存グラフ，8 つの前提条件

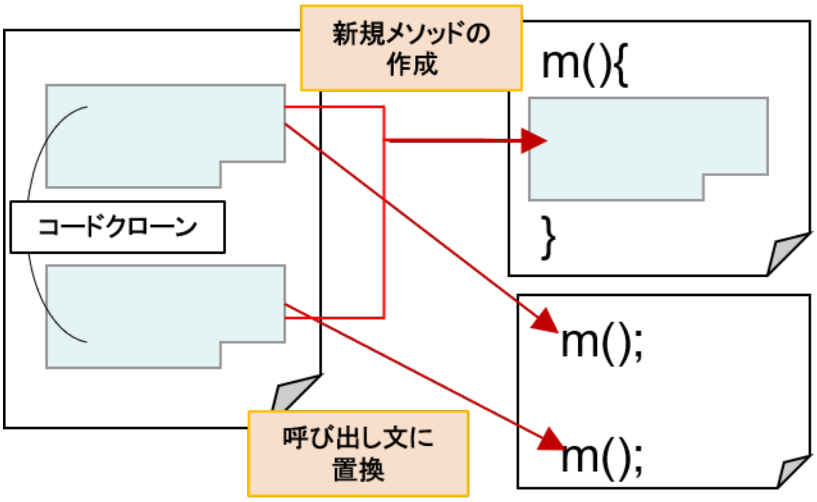

図 2: コードクローンのリファクタリング

によりリファクタリング可能性を判定する．ネスト構造木やプログラム依存グラフの比較によって前提条件が満たされているかを分析し，すべての前提条件満たしている場合のみ，リファクタリング可能なクローンペアとして判定する．なお，JDeodorant のリファクタリング可能性判定はタイプ 1 からタイプ 3 のコードクローンに対応可能である．

以下では，JDeodorant のリファクタリング可能性判定における前提条件や問題点について述べる．

2.2.1　リファクタリング可能性

Tsantalis らはリファクタリング可能性判定のための 8 つの前提条件を提示している．クローンペアのリファクタリング可能性はそのクローンペアが前提条件のすべて満たした場合のみ，リファクタリング可能と判定され，1 つでも違反した場合にはリファクタリング困難と判定される.8 つの前提条件を以下に示す．

1. 変数のパラメータ化の際に既存の制御やデータの依存関係，出力の振る舞いに変更があってはならない
2. それぞれ異なるサブクラス型をもつ変数は、共通のスーパークラスで宣言されているメソッドか、あるいはオーバーライドされたメソッドのみを呼び出す必要がある
3. フィールド変数のパラメータ化はその値が変更されていない時のみ可能である

4. メソッド呼び出しのパラメータ化は void 型を返さない時のみ可能である
5. 対応してないステートメントは既存の制御やデータの依存関係，出力の振る舞いに変更なく、クローンの前後に移動できなければならない
6. 抽出されたメソッドは同じ型の変数をもとのメソッドに返さなければならない
7. 条件付き return 文が含まれてはならない
8. 分岐処理の命令文（BREAK,CONTINUE）があれば、それに対応する反復命令文が含まれていなければならない

JDeodorant のリファクタリング可能性判定の結果は Excel ファイルに出力される．Excel ファイルにはクローンセット内のコードクローンについての情報とクローンペアのリファクタリング可能性が出力される．図 3 に出力される Excel ファイルの内容の一部を表す．
図 3 において左側にクローン ID やファイル、メソッド、開始行、クローン数などコードクローンの情報がテキストで出力される．コードクローンの情報は行によってクローンセットの何番目のコードクローンであるかを示しており，例えば，1 行目のセルには 1 番目のコードクローンの情報が出力されている．
右側にはクローンペアのリファクタリング可能性をセルの色によって示している．赤色もしくは緑色に塗られている部分がクローンペアのリファクタリング可能性を表しており，. 視覚的に判断しやすく表示される．色が塗られてない箇所にあたるクローンペアはリファクタリング可能性判定が行われていない．これはクローンペアに複数のメソッドが含まれている場合，もしくは，なんらかの不具合が JDeoedorant の処理で生じた場合とされている．リファクタリング可能性は行と列によってクローンセットにおけるどの組み合わせのクローンペアのリファクタリング可能性なのかを示す．例えば，1 行目 1 列目ならば 1 番と 2 番，1 行目 2 列目ならば 1 番と 3 番，2 行目 1 列目なら 2 番と 3 番の組み合わせのクローンペアのリファクタリング可能性を示している．
また，リファクタリング可能性を表すセルには JDeodorant によって生成された html ファイルへのリンクが含まれている.html ファイルではネスト構造木のサブツリーの数やクローンペアの位置関係，検出時の経過時間など，リファクタリング可能性判定の詳細な情報が確認できる. リファクタリング可能性が困難と判定されている場合はなぜ困難判定を受けたのかの理由も簡単に記されている．

本研究においてはリファクタリング可能性判定結果の比較に使用するほかにコードクローンの位置を特定するためにコードクローンの開始行と終了行の情報を利用した．

677	src	gr.uom.jav	Statement			33	95	1353	3505			2	Clones are within the		677-1-2		
677	src	gr.uom.jav	Statement			38	100	1530	3687								
733	src	gr.uom.jav	ExtractClo	extractClo	()V	752	763	43942	44908	608	9	2	Clones are	0	733-1-2		
733	src	gr.uom.jav	ExtractClo	copyConst	(QMethod	1619	1630	100817	101624	28	9						
734	src	gr.uom.jav	ReplaceCc	modifyInh	()V	451	466	25892	27054	459	11	4	Clones are	0	734-1-2	734-1-3	734-1-4
734	src	gr.uom.jav	ReplaceTy	createInte	(QList<QS	1817	1832	114112	115332	340	11				734-2-3	734-2-4	
734	src	gr.uom.jav	ReplaceTy	createStat	()V	1136	1151	70090	71293	502	11				734-3-4		
734	src	gr.uom.jav	ReplaceTy	createStat	()V	1518	1533	94153	95307	502	11						
735	src	gr.uom.jav	ReplaceTy	createInte	(QList<QS	1817	1846	114112	116984	340	24	3	Clones are	0	735-1-2	735-1-3	
735	src	gr.uom.jav	ReplaceCc	modifyInh	()V	672	702	38890	41596	459	24				735-2-3		
735	src	gr.uom.jav	ReplaceTy	createStat	()V	1518	1547	94153	96892	502	24						
738	src	gr.uom.jav	ExtractCla	createSett	(QIVariabl	1268	1274	72867	73575	40	5	2	Clones are	0	738-1-2		
738	src	gr.uom.jav	ExtractCla	createGett	(QIVariabl	1317	1323	76695	77384	29	5						
757	src	gr.uom.jav	CFGTryNo			31	36	945	1213			2	Clones are in differe		757-1-2		
757	src	gr.uom.jav	PDGTryNo			21	26	770	1019								
767	src	gr.uom.jav	ExtractCla	modifySou	(QMethod	1962	1967	116672	117118	612	5	4	Clones are	0	767-1-2	767-1-3	767-1-4
767	src	gr.uom.jav	ExtractCla	modifySou	(QMethod	2019	2024	120254	120700	612	5				767-2-3	767-2-4	
767	src	gr.uom.jav	ExtractCla	modifySou	(QMethod	2230	2235	134071	134485	612	5				767-3-4		
767	src	gr.uom.jav	ExtractCla	modifySou	(QMethod	2210	2215	132737	133154	612	5						

図 3: リファクタリング可能性判定の出力 Excel ファイルの一部

2.2.2 リファクタリング可能性の検出における問題点

JDeodorannt が行うリファクタリング可能性判定には以下のような問題が確認される．

複雑でない作業でリファクタリング可能なクローンペアに対するリファクタリング困難判定

> JDeodorant がリファクタリング困難と判定するクローンペアはリファクタリングを行うために複雑な作業が要求されるなど大きなコストがかかりうるはずである．しかし，リファクタリング困難と判定されたクローンペアの一部には複雑でないリファクタリングで集約可能であるものが含まれていたことが確認された．すなわち，一部のクローンペアに対してはリファクタリング可能性を誤ってリファクタリング困難と判定してしまう．

OSS へのリファクタリング可能性判定の際に発見した，リファクタリング可能性が誤って判定されている例を以下に示す．
図 4 に記した 3 つのコード片はそれぞれ他の 2 つのコード片とクローンペアとなっている．これらのコードクローンは互いにタイプ 2 に分類されるコードクローンであり，ハイライト部分であるメソッドの引数の一部のみを除いて完全に一致している．これらのコードクローンからなるクローンペアの組み合わせのうち，(a) と (c) のコードクローンのクローンペアのみリファクタリング困難と判定され，ほかの組み合わせはリファクタリング可能と判定される．

JDeodorant の判定結果によるとハイライトの箇所でフィールド呼び出しを行っている MethodInvocation と ReturnStatement の共通の親クラスが同名のフィールドである EXPRESSION_PROPERTY を持たないことがリファクタリング困難と判定した理由になって

(a)

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
sourceClassParameter =
addSourceClassParameterToMovedMethod(newMethodDeclaration, targetRewriter);
addThisVariable(additionalArgumentsAddedToMovedMethod);
additionalParametersAddedToMovedMethod.add(sourceClassParameter);
}
targetRewriter.set(newMethodInvocation, MethodInvocation.EXPRESSION_PROPERTY,
parameterName, null);
```

(b)

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
sourceClassParameter =
addSourceClassParameterToMovedMethod(newMethodDeclaration, targetRewriter);
addThisVariable(additionalArgumentsAddedToMovedMethod);
additionalParametersAddedToMovedMethod.add(sourceClassParameter);
}
targetRewriter.set(fragment, VariableDeclarationFragment.INITIALIZER_PROPERTY,
parameterName, null);
```

(c)

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
sourceClassParameter =
addSourceClassParameterToMovedMethod(newMethodDeclaration, targetRewriter);
addThisVariable(additionalArgumentsAddedToMovedMethod);
additionalParametersAddedToMovedMethod.add(sourceClassParameter);
}
targetRewriter.set(newReturnStatement, ReturnStatement.EXPRESSION_PROPERTY,
parameterName, null);
```

図 4: 誤ったリファクタリング可能性判定が行われるクローンペアの例

いる．つまり，JDeodorant はこういったクローンペアに対しては親クラスへの集約を試すこと分かる. 図 5 は親クラスの集約によるリファクタリングを表す．クラス A，B はクラス Parent と親子関係にあり，クローンペア内でそれぞれ A，B がフィールド f を呼び出している．親クラスへの集約ではクラス A，B の親クラスであるクラス Parent を引数としたメソッドを作成し，メソッドの呼び出しの際に A、B を引数として渡す．ただし，作成されるメソッド内で親クラスが，コードクローン内にて子クラスの呼び出していたフィールドやメソッドを呼び出す必要があり，. 図 5 のようにクラス Parent がフィールド f を持たないといった場合には，この集約は不可能である．すなわち，図 4 の (a) と (c) のクローンペアにおいても，MethodInvocation と ReturnStatement を共通の親クラスに集約しようとするが，親クラスがフィールド EXPRESSION_PROPERTY を持たないために，親クラスへの集約が不可能だと判断し，その時点で JDeodorant はリファクタリングが不可能だと判定してしまう．

しかし，実際には (a) と (c) のクローンペアにおいても，(a) と (b) や (b) と (c) のクローンペアに行われるような「フィールド呼び出しを行う引数の導入」により，リファクタリングが可能である．. 図 6 は「フィールド呼び出しを行う引数の導入」によるリファクタリング

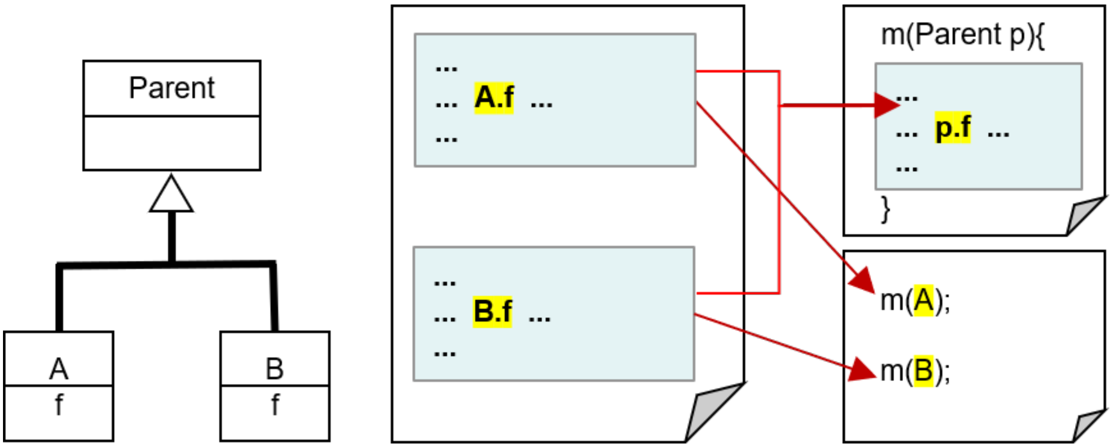

図 5: 親クラスへの集約

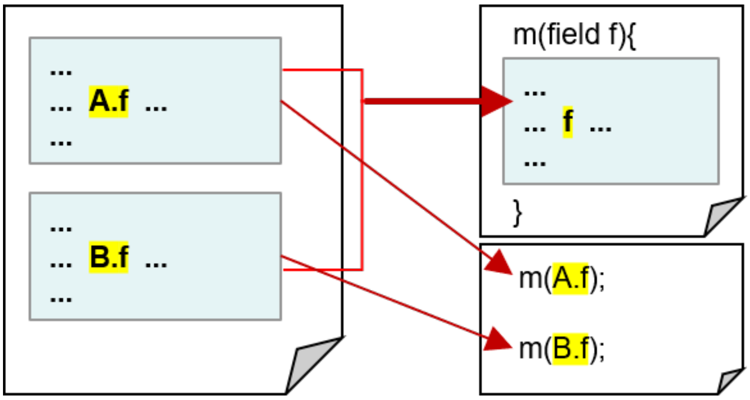

図 6: フィールド呼び出しを行う引数の導入

を表す．このリファクタリングは親クラスの集約より単純で，フィールド f を引数としたメソッドを作成し，メソッドの呼び出しの際には AB が呼び出したフィールド f を引数として渡す．ただし，このリファクタリングは 2.2.1 節の前提条件「フィールド変数のパラメータ化はその値が変更されていない時のみ可能である」を満たしている必要がある．図 4 のコードクローンにおいてフィールド変数 EXPRESSION_PROPERTY は値が変更されてなく，問題なくリファクタリングできる．

そのため，図 4 の (a) と (c) のクローンペアは実際には複雑な作業を行わずにリファクタリングが可能であるのに，コードクローン内で呼び出すフィールドにより，既存手法においてリファクタリング困難と判定されてしまう．本研究では，既存手法である JDeodorant のリファクタリング可能性判定におけるこのような誤判定を軽減する手法を提案する．

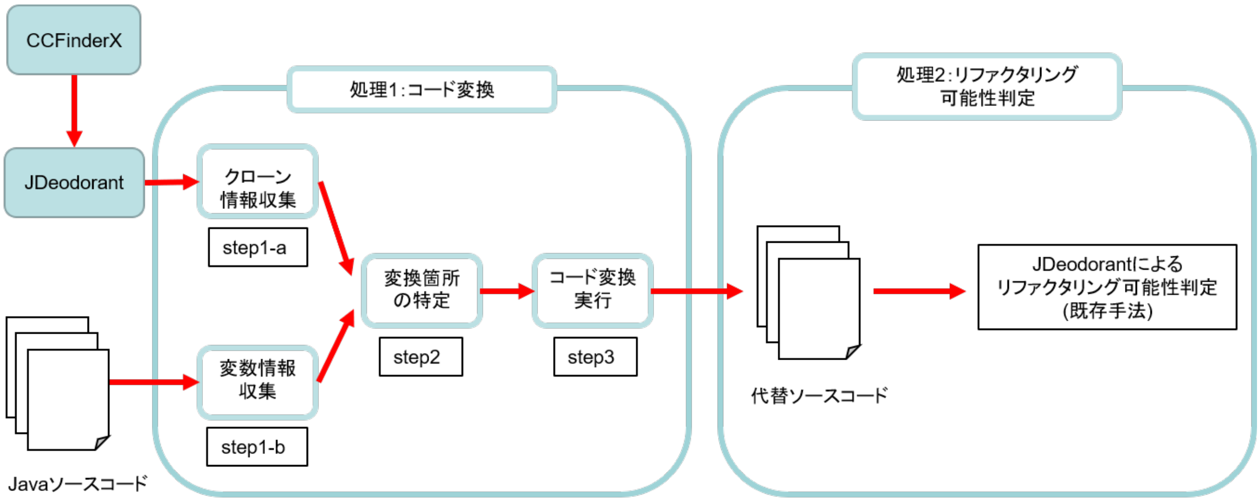

図 7: 提案手法の概要

3　提案手法

本章では，提案手法の処理について説明する．本手法では，次のように大きく 2 つに分けられるフェイズによってリファクタリング可能性判定を行う．

フェイズ 1： 対象となるソースコードに対してのコード変換を行う．

フェイズ 2： コード変換により生成される代替コードに対し，既存のリファクタリング可能性判定を行う．

図 7 はフェイズの流れを示す．

なお，JDeodorant によるリファクタリング可能性判定は Java 言語のソースコードが対象であるため，本手法においても Java 言語のソースコードを対象とする．また，フェイズ 2 における既存のリファクタリング可能性判定については参考文献 [3] にて詳しく解説されているため，本論文では説明を省く．

よって，以降の節では提案する手法のフェイズ 1 におけるコード変換の各ステップの詳細について述べる．3.1 節から 3.4 節の処理をプロジェクト内のすべてのソースファイルに対して行うことでコード変換を完了させ，コード変換適用後のソースファイルに対して既存のリファクタリング可能性判定を行うことで提案する判定手法を実現する．ただし，Java ソースファイルの判断には拡張子によって行うため，実装の都合などで拡張子が.java でない java ソースファイルにはコード変換を行えない．

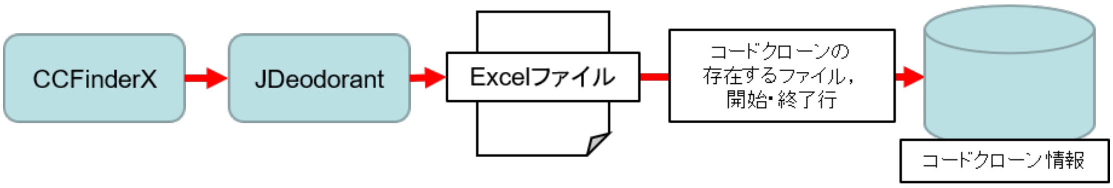

図 8: ステップ 1-a の処理

3.1 ステップ 1-a:クローン情報収集

ステップ 1-a ではコードクローンの範囲を調べる．4.1 節での説明の通り，コードクローン内のフィールド変数の呼び出しがリファクタリング可能性判定が誤りを起こす原因となる．そのため，コード変換を行うためにコードクローンに含まれているフィールド変数呼び出しを探す必要があり，フィールド呼び出しがコードクローンに含まれているかを判別ができるように，コードクローンの位置が必要となる．そのため，図 8 のようにコードクローンの位置情報を集める．

JDeodorant は複数の種類のコードクローン検出ツールに対応しているが，コード変換をそれぞれのツールの出力に対応したものにするのは難しく開発の時間も足りなかった．JDeodorant はクローン検出ツールにより検出されたコードクローンの情報を Excel ファイルに出力する.(2.2.1 節・図 3) 本手法ではこの Excel ファイルからソースコードのコードクローンの情報を取得した．評価実験においては CCFinderX の出力を用いてコードクローンの位置情報を取得したが，他のコードクローン検出ツールでも JDeodorant によりコードクローンの情報は出力されるため，対応できるはずである．

出力 Excel ファイルにはコードクローンを含んでるソースコードのソースフォルダー，パッケージ，クラス名やコードクローンの開始行と終了行などが含まれているが，コードクローンの存在位置が必要であるため，コードクローンの存在するソースファイルの名前と位置，コードクローンの開始行と終了行を取得する．

3.2 ステップ 1-b:変数情報収集

コード変換については詳しくは 3.4 節で説明するが，コード変換のためにソースコード内の変数の情報や呼び出しているフィールド変数の情報が必要となる．よって，ステップ 1-b では図 9 のように変数・フィールド変数の情報を調べる．
コード変換の対象となるソースコードに対して構文解析を行い，変数宣言箇所を特定する．変数宣言文やその位置から，ソースコード内の変数の名前や型，スコープをリスト化する．また，呼び出される可能性のあるフィールドの情報を得るために，コード変換の対象となる

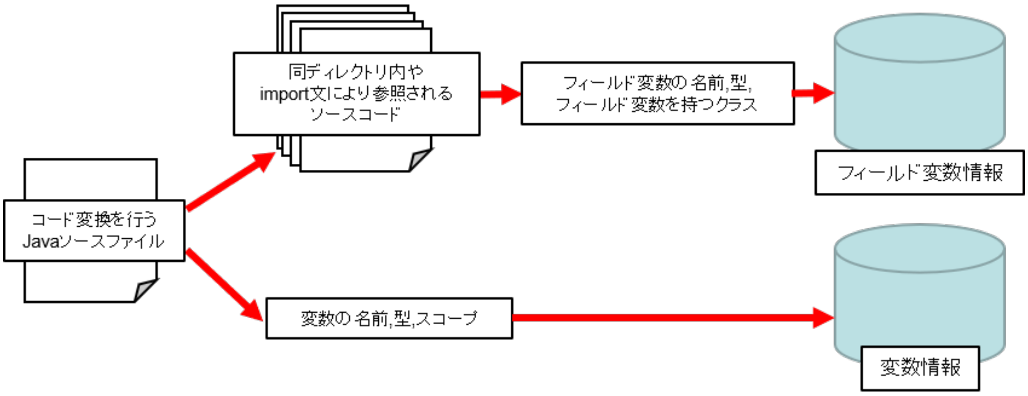

図 9: ステップ 1-b の処理

ソースコードと同じディレクトリ内にあるソースコードや，ソースコード内で import 文で宣言された参照される可能性のあるクラスに対しても構文解析を行い，フィールド変数の名前と型，そのフィールド変数を持つクラス名をリスト化する．

3.3 ステップ 2:変換箇所特定

ステップ 2 ではコード変換を行うべき場所の特定を行う．まず，JDeodorant のリファクタリング可能性判定において誤判定が起きる原因をまとめる．
4.1 節から分かるようにリファクタリング可能性判定を受けるクローンペアにおいて，

1. クローンペアの相違点がコードクローン内でフィールド呼び出しを行う子クラス型変数の型である．
2. その共通の親クラスに同名フィールドがなく親クラスへの集約が行えない．
3. 上記の 2 点を同時に満たす際に親クラスへの集約以外の集約の可能性を探らずにリファクタリング困難として判定を終わらせてしまう．

というのがリファクタリング困難と誤って判定してしまう原因である. そのため，コード変換はコードクローン内のフィールド呼び出しに対して行う.(詳しくは 3.4 節で説明する)
ただし，コードクローン内のフィールド変数呼び出しのうち呼び出されているフィールド変数の型が分かる場合しかコード変換は実行できない．ソースコードに対して構文解析を行い，ソースコード内のフィールド変数呼び出し $call_f$ が以下を満たしているを確かめ，満たしてる場合のみコード変換を実行する．

$$\{p_{begin}(c) \leq p_{begin}(call_f) < p_{end}() \leq p_{end}(c) | c \in S_{1a}\} \quad (1)$$

$$\{call_f \rightarrow f | f \in S_{1b}\} \quad (2)$$

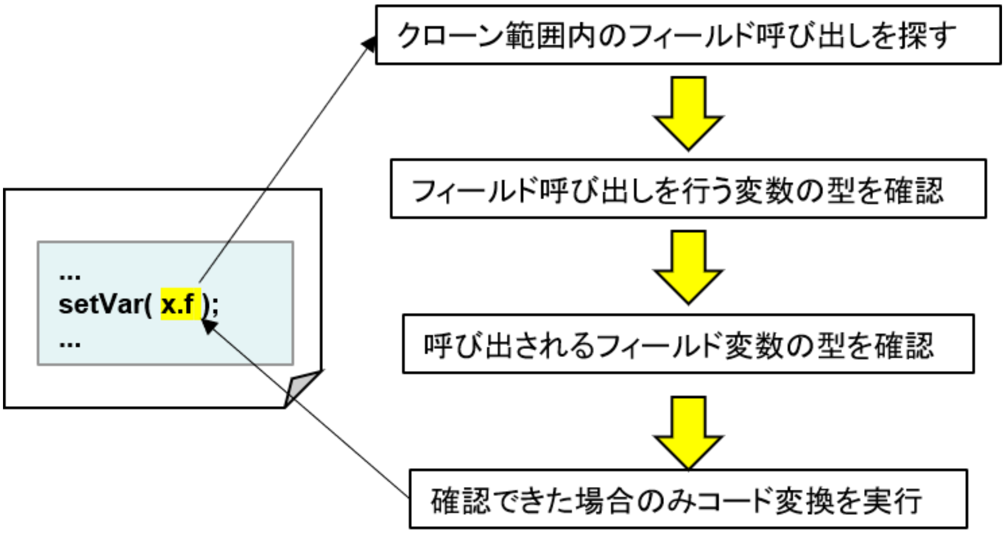

図 10: ステップ 2 の処理

式 (1) について c はクローンを表しており，p_{begin}，p_{end} はコードクローンやフィールド変数呼び出しの開始位置・終了位置を表している.S_{1a} はステップ 1-a で位置を特定できたコードクローンの集合を示す. 式 (1) では，コードクローンとフィールド変数呼び出しの包含関係を示しており，フィールド変数呼び出しがコードクローンに含まれている場合に限定しする. これはフィールド変数呼び出しがコードクローン内に存在する場合のみ，リファクタリング可能性判定の誤りを引き起こす原因になりえるからである.

式 (2) について $call_f \to f$ はフィールド変数呼び出し $call_f$ によってフィールド変数 f を呼び出していることを示す.S_{1b} はステップ 1-b で型の特定できたフィールド変数の集合を表す. フィールド変数呼び出しを新規の変数で宣言するため，呼び出される変数の型が分からなければならない．本手法では，プロジェクトが参照するディレクトリに含まれている Java ソースコードはすべて調べるが，ソースコードがディレクトリの外に存在する，jar ファイルなどのライブラリに含まれるクラスのフィールド変数を呼び出しているなどの理由で，型の特定ができないフィールド変数も存在しうる. そのため，そういった変数を呼び出す箇所についてはコード変換を行わない.

ステップ 2 の処理を図 10 にまとめる. まず構文解析とステップ 1-a の結果からコードクローンに含まれるフィールド呼び出しを探す. そのフィールドを呼び出している変数の型やフィールドの型がステップ 1-b に集めた情報から特定できるか確かめる. これらが確認できた場合のみそのフィールド呼び出しに対してコード変換を実行する,

3.4 ステップ 3:コード変換実行

ステップ 3 ではコード変換を行う．4.1 節，3.3 節からもわかるようにコードクローンがフィールド呼び出しを含んでる際に誤判定が起きる場合がある．よって，コード変換によっ

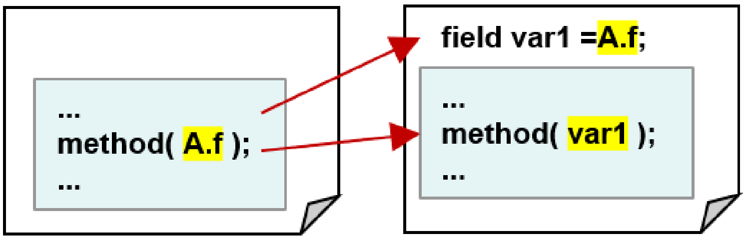

図 11: ステップ 3 の処理

てコードクローン内のフィールド変数呼び出しをコードクローンの外へと移動し，誤判定の発生を避ける．ステップ 2 の時点でコード変換を適用すべき箇所，その箇所を含むクローンの範囲，呼び出しているフィールド変数の型が分かっており，これらを活用しコード変換を実行する．ステップ 2 で特定したコード変換を適用すべき箇所に対し，次の 2 つの変更を行い，コード変換を行う．

1. 新規の変数をコードクローンの外で宣言し，対象となっているフィールド呼び出しの値を代入する．

2. フィールド呼び出しを行っている箇所を新規作成した変数に置き換える．

ステップ 3 の処理を図 11 に示す．

四角で囲まれた箇所がステップ 1-a にて特定したコードクローンの範囲であり，左のコードのハイライトの箇所がステップ 2 にてコード変換を行うと特定した箇所であり，コードクローンの内部でオブジェクト A がフィールド変数 f を呼び出している．この箇所が既存手法において誤判定を引き起こす可能性がある．そのためコード変換を適用することで，誤判定を避ける

まず，1. の変更を適用し，コードクローンの外で新しい $field$ 型の変数 $var1$ を宣言し，A が呼び出しているフィールド変数 f を代入する．そして，2. の変更を適用し，コードクローン内のフィールド呼び出し箇所である $A.f$ を新規の変数 $var1$ に置き換えている．

このコード変換をステップ 2 で特定したすべての箇所に対して行うことで代替コードを作成する．

ちなみに代替コードとコード変換前のコードが同じ動作をし，フィールド呼び出しを行う引数の導入を用いたリファクタリングが行えるのは 2.2.1 節の前提条件「フィールド変数のパラメータ化はその値が変更されていない時のみ可能である」を満たしている必要がある．すなわち，呼び出されているフィールド変数の値がコードクローン内で変更されてないときのみ，代替コードはコード変換前のコードと同等の動作をする．そうでない場合，コード変換の前後のコードの動作は基本的に同等にはならない．ただし，フィールド変数の値の変更を検知するための処理を考案・追加する時間がなかったので，本手法においてリファクタリ

ング可能性判定を行う対象となるクローンペアはコードクローン内でのフィールドの値の変更が行われていないものに限定しているものとし，その条件を満たさないクローンペアに対して動作を行うことを想定しない.

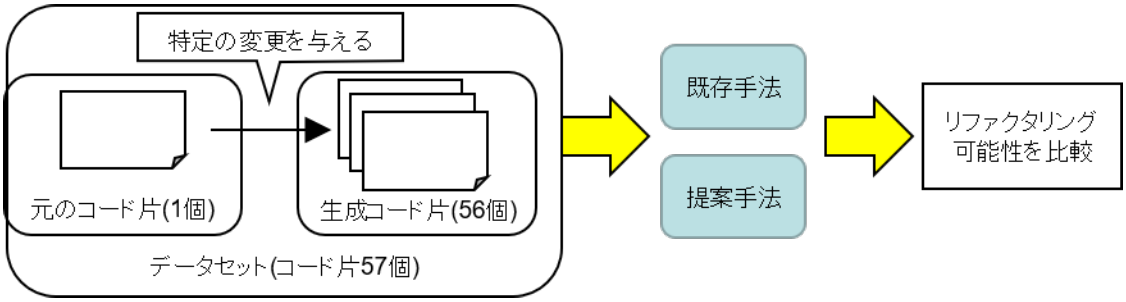

図 12: 評価実験の概要

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
sourceClassParameter = addSourceClassParameterToMovedMethod(newMethodDeclaration,
targetRewriter);
addThisVariable(additionalArgumentsAddedToMovedMethod);
additionalParametersAddedToMovedMethod.add(sourceClassParameter);
}
targetRewriter.set(newMethodInvocation, MethodInvocation.EXPRESSION_PROPERTY,
parameterName, null);
```

図 13: 変更元のコード片

4　評価実験

本章では，評価実験について説明する。図 12 に評価実験の概要を載せる．OSS より既存手法において誤判定の起きる可能性のあるコード片を 1 つ抽出する．そのコード片に対して複数種類の変更の組み合わせを与えることでコード片の集合を生成する．生成したコード片の集合に変更前のコードを加えたものをデータセットとする．データセットに対して，既存手法と提案手法の両方のリファクタリング可能性判定を行い，結果を比較する．

4.1　データセット

データセットは OSS より抽出したコード片の 1 つをもとに作成した．変更を行うコード片を図 13 に示されているもので，OSS である JDeodorant 自身のソースコードから抽出したもので，図 4 に乗せたコードクローンの 1 つである．節でも述べた通り，このコード片を含むクローンペアのリファクタリング可能性が既存手法において誤ってリファクタリング困難と判定され，コード片の内部でフィールド変数が変更されることはないことが分かっている．

図 13 のコード片に対して次のような変更を行う.

A. フィールドを呼び出している変数の型や名前を変更する. 次のいずれかの変更を行う.

  1. 変数の名前を変更する.

  2. 変数の型を共通のスーパークラスを持つほかのサブクラスに変更する.

  3. 1. と 2. の変更を同時に行う.

B. 呼び出されるフィールド変数について変更する. 次のいずれかの変更を行う.

  1. 別名のフィールド変数の呼び出しに変更する.

  2. 別の型を持つフィールド変数の呼び出しに変更する.

C. 行の追加・削除によって変更する. 次のいずれかの変更を行う.

  1. 文を 1 行追加する.

  2. 文を 1 行削除する.

  3 文を別の文に書き換える..

A〜C の変更の一部を図に載せる.
図 14 は変更 A3 によって変更されたコードである. フィールド呼び出しを行う変数 MethodInvocation が別名・別型の変数 ReturnStatement に変更されている.
図 15 は変更 B1 によって変更されたコードである. 呼び出されるフィールド EXPRESSION_ PROPERTY が同じ型の別名フィールド NAME_ PROPERTY に変更されている.
図 16 は変更 C3 によって変更されたコードである. クローン中の additionalParametersAddedToMovedMethod によるメソッド呼び出し文が変数 Y の値を増やす文に変更されている.
ただし，変更によって生成されたコード片がプロジェクト内でエラーを起こす場合，JDeodorant はリファクタリング可能性判定を行うことができない. そのため，生成されたコード片によってエラーが起きることは避けなければならない. 変更 B2 であれば，生成されるコード片がエラーを起こさないようにするため，変更前後のどちらの型でも対応できる箇所，例えば，複数種類持つメソッドの引数などに当たる箇所の，フィールド呼び出しに対して変更は行われる必要性がある.
また，図 4 のコード片と同様，生成されたコードはコード内でフィールドの変更が行われていない.

4.2　実験結果

まず評価実験において元となったコード片と，変更を加えてできたコード片とのリファクタリング可能性判定の結果を表 1 にまとめる．表における数字はどの変更を行ったかを示しており，B の欄の数字が 2 であれば変更 B2 が行われていることを示す. また数字が 0 の場合その変更が行われていないことを示している．/で区切られた複数の数字の場合はそれらのどの数字が示してる変更であっても同じ結果が得られたことを示している. 例えば A，B，C の欄の数字がそれぞれ 0/1，2，0 である場合,「変更 B2 のみを適用した結果」かつ「変更 A1 と変更 B2 を適用した結果」を示している．既存手法と提案手法の欄にはそれぞれのリファクタリング可能性判定における判定結果である. リファクタリング可能は可能，リファクタリング困難は困難と表記している. 判定不可の表記の部分では，その変更により生成されるコード片と元のコード片がクローンペアと判定されなかったことを示している．評価の欄には判定結果がどう変化したかを示している. ◎は既存手法においてリファクタリング困難であり提案手法により改善されたことを示す. ○はどちらの手法でもリファクタリング可能と判定され問題のないことを示す. ×は既存手法においてリファクタリング困難と判定，もしくは，クローンペアと判定されなかったにもかかわらず，既存手法においても改善が見られなかったことを示す. ちなみに，既存手法でリファクタリング可能と判定され，提案手法でリファクタリング困難と判定された，というような悪化することはこの評価実験では存在しなかった.

A	B	C	既存手法	提案手法	評価
0/1	0/1	0	可能	可能	○
0/1	2	0	困難	困難	×
2/3	0	0	困難	可能	◎
2/3	1	0	可能	可能	○
2/3	2	0	困難	困難	×
0/1/2/3	0/1/2	1/2/3	判定不可	判定不可	×

表 1: コード片に与えた修正とそれにより生成されるコード片とのリファクタリング可能性

評価対象に含まれるリファクタリング可能と判定されたクローンペアの割合を次の表 2 にまとめる.CCFinder により検出されたクローンペアの総数は 412 組であった. ちなみにこれらのコードクローンは「フィールド呼び出しを行う引数の導入」を利用することでリファクタリング可能であり，すべてのクローンペアがリファクタリング可能の判定を受けるのが理想である.

データセットのコード片　57 個：クローンペア数: 412 組	既存手法	提案手法
リファクタリング可能と判定されたクローンペアの数	114	212
リファクタリング可能と判定されたクローンペアの割合	0.28	0.51

表 2: リファクタリング可能なクローンペアの数とクローンペア全体における割合

表 1 から分かるように変更 C1，C2，C3 を加えたコード片と元のコード片の組はクローンペアとして検出されなかった. 行の追加・削除によりタイプ 3 のコードクローンとなったことが原因だと考えられる.

また，表 2 におけるリファクタリング可能なクローンペアは増えているが，表 1 より既存手法で正しく判定できるクローンペアはすべて提案手法でも正しく判定されているため，98 組のクローンペアが提案手法によって誤判定が解消され，114 組のクローンペアが既存手法・提案手法の両方でリファクタリング可能と判定されたことが確認できる. 表 2 におけるリファクタリング可能なクローンペアも確実に増えていることが分かる.

提案手法によりリファクタリング可能性判定が改善されたクローンペアは表 1 より，フィールド呼び出しを行う変数の型のみが異なり，呼び出すフィールド変数の名前や型は同じクローンペアに限られている. コード変換により，コードクローン内のフィールド呼び出しをただの変数として扱うようになるので，フィールド呼び出しが原因の誤判定は解消される. また，誤判定はクローンペアの相違点がフィールド呼び出しを行う変数の型である際に共通の親クラスへの集約を試して失敗した時に発生する. 呼び出されるフィールド変数の名前や型が違うと，そもそも親クラスへの集約を試せない. 以上より，クローンペアにおいて，フィールド呼び出しを行う変数の型のみが異なり，呼び出すフィールド変数の名前や型は同じという，限定された場合のみ判定の改善が行われることが分かる.

その一方で表 1 において，変更 B2 を加えることによって生成されるフィールドの型が異なるコード片と元のコード片の組のクローンペアは提案手法においてもリファクタリング困難の判定を受け，誤判定は残ったままである. JDeodorant の出力によると，このクローンペアはフィールド変数の型によってリファクタリング困難の判定を受けていた. コード変換によってフィールド呼び出しはただの変数に置き換わるが，型の違いは残ったままになる. そのため，コード変換の前後でリファクタリング困難の判定は変わらない. フィールドの型の変更によって生成されたコード片と元のコード片のクローンペアは「フィールド呼び出しを行う引数の導入」に加え，条件式や引数により処理の変わるメソッドを作成することでリファクタリングが可能である.

図 17 にリファクタリング可能性判定の出力結果である Excel ファイルの一部を載せる. 元のコード片とのクローンペアのリファクタリング可能性を示してる箇所であり，上が既存手

法，下が提案手法によるものである．2.2.1 節に記されている図 3 に記される Excel ファイルの右側の部分であり，行と列によりクローンセットの組み合わせを示し、緑と赤によってそれぞれリファクタリング可能とリファクタリング困難の判定を受けたことを示す.(例えば 3 行目 3 列目が緑色のセルであればクローンセットの 3 番目と 6 番目のクローンペアがリファクタリング可能である) 既存手法に比べて提案手法の方が緑の部分が多く，図 17 からも提案手法によって正しく判定されるクローンペアが増えていることが分かる.

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
                    ...
}
targetRewriter.set(newMethodInvocation, MethodInvocation.EXPRESSION_PROPERTY,
parameterName, null);
```

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
                    ...
}
targetRewriter.set(newMethodInvocation, ReturnStatement.EXPRESSION_PROPERTY,
parameterName, null);
```

図 14: 変更 A3 により生成されるコード片

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
                    ...
}
targetRewriter.set(newMethodInvocation, MethodInvocation.EXPRESSION_PROPERTY,
parameterName, null);
```

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
                    ...
}
targetRewriter.set(newMethodInvocation, MethodInvocation. NAME _PROPERTY,
parameterName, null);
```

図 15: 変更 B1 により生成されるコード片

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
                    ...
addThisVariable(additionalArgumentsAddedToMovedMethod);
additionalParametersAddedToMovedMethod.add(sourceClassParameter);
}
...
```

```
if(!containsThisVariable(additionalArgumentsAddedToMovedMethod)) {
                    ...
addThisVariable(additionalArgumentsAddedToMovedMethod);
X++;
}
...
```

図 16: 変更 C3 により生成されるコード片

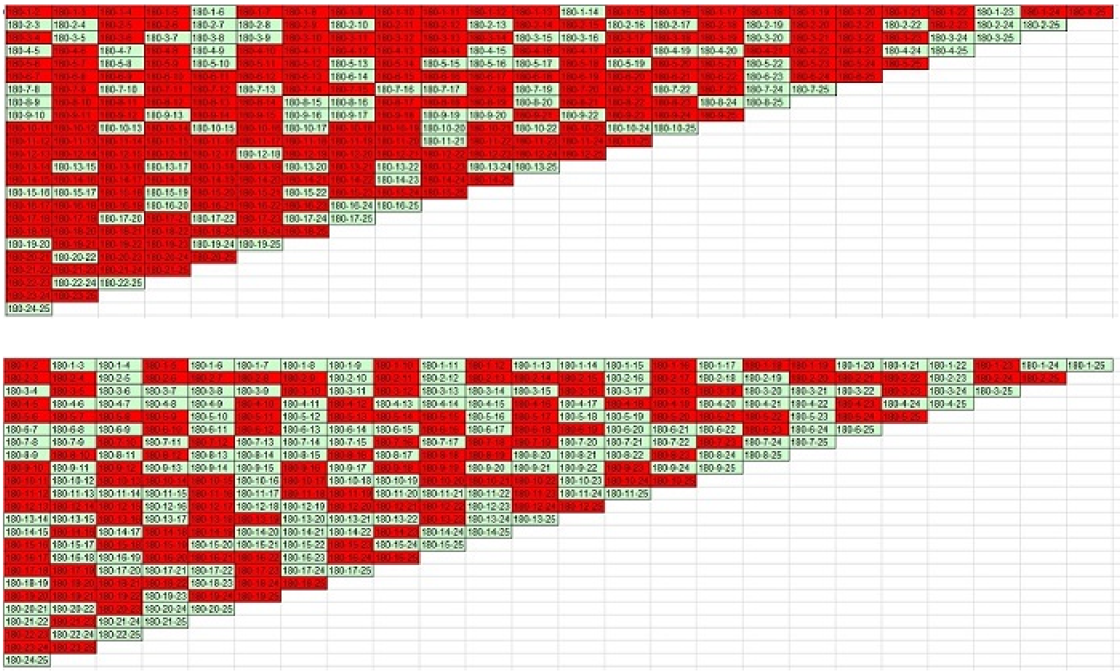

図 17: 出力 Excel ファイル (上:既存手法 下:提案手法)

5 まとめ

本研究では，既存のリファクタリング可能性判定で起きる誤判定を回避するコード変換を既存手法に加える判定手法を提案した．コード変換により誤判定を引き起こす原因となるフィールド変数呼び出しをコードクローンの外へと移動する．
評価実験では，既存手法において誤判定の確認できたソースコードのコード片をもとに作成したデータセットに対し，リファクタリング可能性判定を行った．結果としては判定の改善されたクローンペアの割合は 0.23 と多くはないが，判定が悪化するクローンペアは存在せず，確実に増えることを示した．課題としては以下が挙げられる．

- 本手法で対象とするクローンペアをフィールドの変更が含まれてないものに限定したが，ツールによりフィールドの変更を自動検知してクローンペアを限定せずとも適用できる手法にする．
- 誤判定の起きたクローンペアをより多く調査することで，より効果的なコード変換を考案し，対応できる誤判定を増やすことが見込める．対応できる誤判定を増やすための誤判定が起きるクローンペアの調査と新たなコード変換の考案を行う．

謝辞

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 井上 克郎 教授には，非常に御多忙の中多くのご指導を賜りました．井上 教授の御指導によ り，本論文を完成させることができました. 井上 教授に心より深く感謝いたします.

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 松下 誠 准教授には，本論文や発表における問題点の提示など, 多くの御助言を賜りました. 松下 准教授に心より深く感謝いたします.

名古屋大学 大学院情報学研究科附属組込みシステム研究センター 吉田 則裕 准教授には，日々研究に関する直接の御指導を賜りました. 常に適切な御指導及び御助言を頂いたことにより，本論文を完成させることができました. 吉田 准教授に心より深く感謝いたします.

京都工芸繊維大学 情報工学・人間科学系 テニュアトラック 崔 恩瀞 助教には，研究に関する多くの御助言を頂きました崔 恩瀞 助教に心より深く感謝いたします.

大阪大学大学院情報科学研究科 神田 哲也 特任助教には，研究に関する貴重なご意見を賜りました. 神田 哲也 助教に心より深く感謝いたします.

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 藤原 裕士 氏, 本田 紘貴 氏には, 日々研究や発表について御指導をして頂くなど, 多くの御協力を賜りました. 両氏に心より深く感謝いたします.

最後に, その他様々な御指導, 御助言等を頂い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井上研究室の皆様に心より深く感謝いたします.

参考文献

[1] 井上克郎, 神谷年洋, and 楠本真二. コードクローン検出法. *コンピュータ ソフトウェア*, 18(5):529–536, 2001.

[2] 肥後芳樹 and 吉田則裕. コードクローンを対象としたリファクタリング. *コンピュータソフトウェア*, 28(4):4_43–4_56, 2011.

[3] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, 2015.

[4] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools. *Science of Computer Programming*, 74(7):470–495, 2009.

[5] 肥後芳樹, 楠本真二, and 井上克郎. コードクローン検出とその関連技術. *電子情報通信学会論文誌 D*, 91(6):1465–1481, 2008.

[6] J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.

[7] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE, 1999.

[8] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

[9] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[11] CloneDR. `http://www.semdesigns.com/Products/Clone/`.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[13] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001.

[14] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.