コードクローンの動作を比較するための コードクローン周辺コードの解析

ブャンネメフ オドフー[1,a)]　眞鍋 雄貴[2,b)]　伊達 浩典[1,c)]　石尾 隆[1,d)]　井上 克郎[1,e)]

概要：ソフトウェアの保守を困難にする要因の一つとしてコードクローンが挙げられる．コードクローンとは，ソースコード中に，互いに類似または一致した部分を持つコード片のことである．各コードクローンは，たとえ記述が同一であってもそれらの周辺のコードに依存して異なる動作をする可能性がある．しかしながら，実際にどの程度コードクローンが周辺コードに依存しているかはわかっていない．本研究では，コードクローンと周辺のコードとの依存関係を明らかにするため，コードクローンの周辺コードの量と，周辺コード間の違いについて調査を行った．その結果，多くのコードクローンに周辺コードが存在し，多くのコードクローン間で周辺コードが異なることを確認した．

キーワード：コードクローン，ソフトウェア保守，周辺コード

1. はじめに

コードクローンとは，ソースコードの中に存在する互いに類似または一致したコード片のことである [1]．コードクローンは主に，ソースコードのコピーアンドペーストや同一処理を意図的に繰り返し書くことによって発生する．一般的にコードクローンの存在は，ソフトウェアの保守を困難にすると言われている [1]．例えば，あるコードクローンにバグがある場合，そのコード片のすべてのコードクローンに対して，同様のバグが存在する可能性がある．

一方，すべてのコードクローンが有害というわけではない [2]．また，コードクローンの発生原因となるコピーアンドペーストなどはプログラマの作業を減らし，効率良くソフトウェアを開発するのに役に立つ場合がある．そこで，対処すべきコードクローンを決めるためには各コードクローンを調査する必要がある．

しかしながら，コードクローンの見た目が同じであっても，異なる動作をする場合がある．一方で，見た目が異なっていても同じ動作をしている場合がある．そのため，コードクローンのみを見ても，コードクローンの動作を正確に理解することはできない．

そこで，コードクローンの動作を比較するため，コードクローンの周辺にあるコードに着目する．各コードクローンの動作は，周辺のコードに影響を受ける可能性がある．Lingxao ら [3] はコードクローン間でコードクローン周辺のコードが異なる場合，コードクローンに潜在的なバグが含まれている可能性が高いとしている．また，コードクローンでない部分より，コードクローンである部分の方が安定性が高い [4], [5] とされており，コードクローン自体に変更がなかったとしても，周辺のコードによりコードクローン自体の動作が変わってしまう場合も考えられる．しかし，実際にコードクローンがそのような周辺コードにどの程度依存しているかは明らかではない．

本研究では，コードクローンの周辺コードを解析し，コードクローンの周辺コードへの依存性を調査した．本調査では，コードクローン内の変数の値に影響を与える要素や，制御に影響を与える要素を周辺コードと規定し，周辺コードの量や，コードクローン間の違いについて調査した．その結果，多くのコードクローンに周辺コードは存在し，コードクローン間で周辺コードが異なる場合も多く存在することを確認した．

以降，2 章では背景として，コードクローン，データフロー，抽象構文木について説明する．3 章では，コードクローンの周辺コードへの依存性の調査について，その方法

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565-0871, Japan
[2] 熊本大学 大学院自然科学研究科
Graduate School of Science and Technology, Kumamoto University, Kumamoto, Kumamoto 860-8555, Japan
[a)] b-odkhuu@ist.osaka-u.ac.jp
[b)] y-manabe@cs.kumamoto-u.ac.jp
[c)] h-date@ist.osaka-u.ac.jp
[d)] ishio@ist.osaka-u.ac.jp
[e)] inoue@ist.osaka-u.ac.jp

と結果について述べる．4 章では関連研究を，5 章ではまとめと今後の課題について述べる．

2. 背景

2.1 コードクローン

コードクローン (Code clone) 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を指す [1]．一般的に，コードクローンの存在はソフトウェアの保守を困難にすると言われている．例えば，コードクローンとなっているコード片中に欠陥（バグ）が存在する場合，そのコード片と一致または類似した他のコードクローンについても同様の欠陥が存在する可能性がある．ソフトウェアのスケールが大きく，コードクローンとなるコード片が多く存在する場合，すべてのコードクローンを把握し，それらのコードクローンに含まれるバグを修正するのは困難である．

コードクローンにはテキストベース，トークンベースなど比較する対象によって様々な検出手法が存在しており，検出手法ごとにコードクローンについての定義が異なる．Bellon ら [6] はコードクローン間の違いの度合いに基づき，コードクローンを以下の 3 つの定義に分類している．

タイプ 1　空白やタブの有無，括弧の位置などのコーディングスタイルを除き，完全に一致するコードクローン．

タイプ 2　変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるコードクローン．

タイプ 3　タイプ 2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン．

本研究ではタイプ 2 のコードクローンを分析対象とする．互いに一致または類似したコードクローンの対をクローンペア (Clone pair) と呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合をクローンセット (Clone set) と呼ぶ．

2.1.1 発生原因

コードクローンの発生原因として以下のものが挙げられる [7]．

既存コードのコピーによる再利用　新しい機能を実現する際，プログラマは似たような動作をする既存コードをコピーしてそのまま，もしくは一部修正して実現することが多い．

コーディングスタイル　コーディングスタイルであるエラー出力やユーザインターフェース表示などのスタイルを保持するために，コード片を意図的にコピーして利用する．

定型処理　多数に繰り返し書く処理は，定型のようになり，プログラマが意図的にコピーしていなくても，同じようなコードになってしまう．

データ構造の違い　同じ型を持つデータ構造の処理を他のデータ構造に使ってしまう．

パフォーマンス改善　厳しい時間制約を持つシステムにおいて，コンパイラがインライン展開を提供していない場合に，最適化するために繰り返し処理を記述することがある．

偶然　偶然に，プログラマが，類似したコードを書いてしまう場合がある．

2.1.2 問題点

ソフトウェアの保守に関する研究は多数のプログラムがかなりの量のコードクローンを含むことを示している．このようなコードクローンは以下の 2 つの理由により有害と判断されている [8], [9]．

(1) 重複している不必要なコードはソフトウェアの保守のコストを上げる．

(2) コードクローンに対する一貫していない変更がバグを作る可能性があり，プログラムの不正確な動作の原因になりうる．

集約などコードクローンを除去し，ソフトウェアの保守性を向上させる手法は数多く提案されている [10], [11]．集約とは，ソースコードの中に数多く存在する同じ処理を行う類似したコード片を 1 つのメソッドにまとめることである．集約によって，ソースコードの量を減らし，よりわかりやすいコードにすることができる．一方，ソフトウェアのソースコードの類似は固有のものもあり，集約，除去などは常に好ましいわけではない．これには以下の 3 つ理由がある．

(1) **プログラミング言語の機能不足**：コードクローンが長い時間にかけて多くの変更によって変化することがある．そのようなコードは簡単には集約されない．

(2) **パフォーマンス関連**：コードクローンを集約したコードはより悪いパフォーマンスを見せる場合がある．

(3) **ソフトウェア開発用**：一部のテスト用コードが集約されるべきでない場合がある．

そのため，すべてのコードクローンを除去できるわけではなく開発者はコードクローンに対する集約の可否を調査する必要がある．

2.1.3 検出ツール

コードクローン検出手法には，ソースコードの字句解析に基づく手法 [12], [13], [14] や，特徴メトリクスに基づく手法 [15], [16] などが存在する．ソースコードの字句解析に基づく手法では，ソースコード中で同一の文字列を検索することでコードクローンの検出を行う．特徴メトリクスに基づく手法では，クラスや関数，ファイルのようなプログラム中のある種の単位ごとに特徴メトリクスを定義・算出し，それらのメトリクス値が類似したものをコードクローンとして抽出する．

本研究ではコードクローン検出システムとして CCFinder[17] を使用する．CCFinder は字句解析ベースのコードクローン検出手法でタイプ 1，タイプ 2 のコード

クローンを検出できる．CCFinder のコードクローンの検出処理は，字句解析，変換処理，検出処理，出力整形処理からなる．字句解析でソースコードをトークン列に変換し，変換処理で変数名や関数名等を同一のトークンに変換する．その後，検出処理で閾値以上の長さの共通トークン列を探索し，すべてのコードクローンの対のリストを出力する．

2.1.4 メトリクス

コードクローンの性質，特徴を表す値として様々なメトリクスが定義されている [17]．本研究では，その中でも RNR というメトリクスを用いる．RNR とはクローンセットに含まれるコード片の非繰り返し度を表すメトリクスであり，値が低いクローンセットは開発者にとって興味がないクローンセットである [18]．

2.1.5 周辺コード

周辺コードに関連する概念として，Lingxiao ら [3] がコンテキストという概念を定義している．Lingxiao らは，あるコード片 F のコンテキストとして，そのコード片 F を囲む最内制御ブロックを指している．最内制御ブロックとはプログラミング言語における制御フロー構造となるものである．例えば，C 言語では，if，switch，for，while，関数宣言などである．

2.2 データフロー

本研究ではコードクローンに含まれる変数についてのデータフローを取り扱うため，Ishio ら [19] の手法を用いる．変数間のデータフロー関係（以降，データフロー関係と呼ぶ）とは代入文の左辺式と右辺式の間に成り立つ関係である．例えば”$p = q$;” という文であれば，q から p へのデータフローがあると考える．また，本研究で扱うデータフロー関係は以下の 4 つのルールに従う．

(1) ある変数 p の値を関数呼び出しで取得する場合，関数呼び出しの引数に変数 q があれば，q から p へのデータフローがあると考える．その関数呼び出しを p，q 間のデータフロー上に出現する関数呼び出しとする．また，ある変数 t から r へのデータフローがあった場合，変数 t によって呼び出される関数もデータフロー上に出現する関数呼び出しになる．

(2) 代入文の右辺式に複数の変数が演算子で結合された場合あるいはメソッドに引数になった場合，それらの変数から代入文の左辺式の変数へのデータフローがあると考える．

(3) データフロー関係は制御の流れを無視する．

(4) ある変数 p と推移的にデータフロー関係がある変数 s があれば，p と q の間にデータフローがあると考える．

図 1 にデータフローの例を示す．図 1 の (a) のプログラムの変数 y に対するデータフローは図 1 の (b) ようになる．

図 1 では，変数 y と 4 つの変数 x，str，a，b はデータフローがある．また，変数 str の呼び出している関数 length はデータフロー上に出現する関数になる．

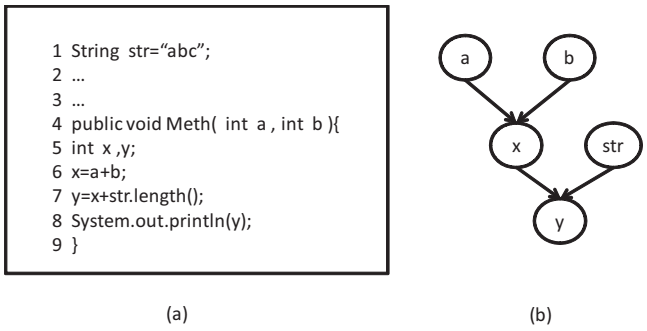

図 1 データフロー図の例，(a) ソースコード，(b)(a) の変数 y に対するデータフロー

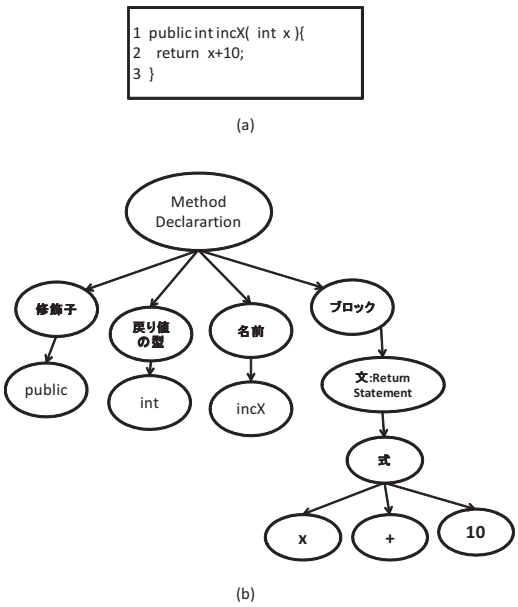

図 2 Eclipse の JDT によって作成される AST のイメージ図

2.3 抽象構文木

抽象構文木（AST，Abstract Syntax Tree）とはプログラムの文法構造を木の形で表したものである．以下に，Java 言語における，Eclipse の JDT によって作成される AST の例を紹介する．**図 2**(a) のコード片に対する，AST の構造は図 2(b) のようになる．図 2(b) で示す各丸が AST の 1 つのノードを表す．親ノードから子ノードへ矢印が引かれている．AST の一番終端のノード（葉）は文字列，リテラルなどの情報を保持し，それより上のノードが式，文，ブロックなどそのノードが根となる部分木が何を表すかを示す．例えば，x というノードから親をたどって行くと，x が $x+10$ という式の一部であり，incX メソッドの “ブロック”に含まれていることがわかる．

3. コードクローンの周辺コードへの依存性調査

コードクローンの周辺コードへの依存性を調査するため，実験を行った．以下，その方法と結果について述べる．

3.1 周辺コード

本調査では，以下の 2 種類を周辺コードとした．

(1) コードクローンの中に存在する変数のデータフローに出現する，フィールド変数，引数を持たないメソッド呼び出し，コードクローンを含むメソッドの仮引数．以降，これらを外部要素と呼ぶ．また，コードクローンの中に存在するある変数のデータフローに出現する外部要素において，外部要素がコードクローンのその変数から参照されているとする．

(2) コードクローンを含む制御ブロックのうち，最も内側にあるブロック．以降，これを最内制御ブロックと呼ぶ．

3.2 リサーチクエスチョン

本調査では以下のリサーチクエスチョンを定めた．

RQ1： 各コードクローンにおいて，周辺コードはどれだけ多いか

RQ2： 各コードクローンの周辺コードは同一のクローンセットの中でどれだけ一致するか

3.3 調査方法

まず CCFinder を用いてコードクローンを検出する．本研究では，検出したコードクローンのうち，1 つのメソッド内に含まれることとコードクローンは必ず変数を含むことの 2 つの条件を満たしたもののみを使用する．なぜなら，これらの条件を満たさないコードクローンは本調査に適さないためである．コードクローンが変数を含まない場合，周辺コードからコードクローンへのデータフローが存在することはない．また，コードクローンが 2 つのメソッドにまたがるときは，最内制御ブロックを特定できない．

次に，各コードクローンについて，コードクローン内の変数を特定する．特定後，同一のクローンセット内にあるコードクローンについて，それらの変数を出現順に対応付ける．

そして，各変数について値の計算に使われている外部要素を特定する．また，抽出した各コードクローンについて，最内制御ブロックを特定する．

各リサーチクエスチョンに答えるために必要なデータは以下のとおりである．

RQ1 クローンセット当たりの外部要素の数と最内ブロックの種類別の頻度

RQ2 1 つのクローンセット内の各コードクローンが参照している同一外部要素の数と，最内制御ブロックの種類

調査対象として用いたプロジェクトは SourceForge.net, tigris.org から取得した，Java で記述されているプロジェクト 7 個である．**表 1** に各プロジェクトのバージョンとソースコードの行数（SCLN）とファイル数を示す．**表 2** にはコードクローンに関する情報である本研究の対象となるクローンセット数（Targeted Clone Set Number，TCSN），コードクローン数（Targeted Code Clone Number，TCCN）と対象とならないコードクローン数（Not Targeted Code Clone Number，NTCCN，変数を含まないコードクローンと 1 つのメソッドに完全に閉じていないコードクローンの数）を示す．

以降，周辺コードの特定方法と，外部要素の数え方，コードクローン内の変数名の一致判定の方法，外部要素への参照の一致判定の方法について述べる．

3.3.1 コードクローンの周辺コードの特定方法

本項では，コードクローンの周辺コードである，外部要素と最内制御ブロックの特定方法について説明する．

外部要素の特定

コードクローンに存在する各変数のデータフロー上に出現する外部要素を取得するために，まずコードクローンに存在するすべての変数のリストを作る．そのリストから変数を 1 つずつ取り出して，データフロー上に出現する外部要素を取得していく．コードクローンの 1 つの変数のデータフローを取得する処理を Algorithm1 に示す．

Algorithm1 は引数としてデータフローを取得したい変数 x をとり外部要素のリストを出力する．変数として結果を

表 1 調査対象プロジェクトのバージョンとソースコードの行数とファイル数

プロジェクト名	バージョン	SCLN	ファイル数
ArgoUML	0.34	109015	1318
DataCrow	3.9.17	68139	650
LaTeXDraw	3.0.0(a4)	35999	363
SweetHome3D	3.7	74352	212
jwebmail	1.0.1	11173	113
jEdit	4.3	103317	502
muCommander	0.8.5	76739	1069

表 2 調査対象プロジェクトから検出したコードクローンに関連する情報

プロジェクト名	TCSN	TCCN	NTCCN
ArgoUML	2073	5276	1227
DataCrow	1337	5540	437
LaTeXDraw	1358	7506	1095
SweetHome3D	2017	8217	766
jwebmail	125	255	21
jEdit	1740	4714	574
muCommander	1496	3386	679

Algorithm 1 変数の外部要素のリストを取得する処理

入力 x : データフローを取得したい変数
出力 *Result* : 入力変数のデータフローに出現する外部要素のリスト

```
1: function getExternalElement(x:variable)
2:     Result = φ;
3:     Stack = {x};
4:     while Stack is not empty do
5:         Stack の末尾の要素を取り出し v とする
6:         変数 v が左辺に出現する代入文すべてを特定する
7:         特定した代入文の右辺に出現する変数，引数なしのメソッドの集合を C とする
8:         C から Result に存在するものを除去する
9:         Result に C の要素をすべて追加
10:        Stack に C の要素をすべて追加
11:    end while
12:    Result からローカル変数を除去する
13:    return Result
14: end function
```

格納する *Result* というリストと作業用のスタック *Stack* を用意し，*Stack* に初期値として x を入れておく．このアルゴリズムでは x を基点に，x に値を代入する文を特定し，その右辺に出現した変数を経由してたどり着けるメソッドの仮引数とフィールド，引数なしのメソッド呼び出しを外部要素として列挙する．

最内制御ブロックの特定

ソースコード上でコードクローンの開始位置から前に存在する制御ブロックを調査し，制御ブロックの範囲がコードクローンの開始位置から終了位置までを完全に含む制御ブロックを最内制御ブロックとする．

3.3.2 外部要素の個数計算

始めに，クローンセット内の各コードクローンの変数から参照されている外部要素を抽出する．次に，各外部要素から情報を取得する．コードクローンを含むメソッドの仮引数からは ⟨ 仮引数，型，出現順序 ⟩，フィールド変数からは ⟨ フィールド，型，名前 ⟩，引数を持たないメソッド呼び出しからは ⟨ メソッド，戻り値の型，名前 ⟩ を取り出す．

最後に，それらの取り出した各情報を要素として持つ集合を作成し，その集合の要素数が外部要素の個数となる．集合を作成することにより同じデータを持つ外部要素が重複することを避けている．

3.3.3 変数名の一致の判定方法

クローンセット内で，変数名が一致する条件は対応がある（以降，対応関係という）変数がすべて同じ名前であり，どのクローンペアにおいてもその条件が成り立つことである．

3.3.4 外部要素への参照の一致判定方法

外部要素への参照の一致判定においては，外部要素が列，コードクローン内の変数が行に対応する表を考える．表の例を**表 3** に示す．あるコードクローン内の変数がある外部要素を直接，もしくは推移的に参照するとき，その対応する行と列の交点にチェック（例の場合は○）を入れていき，すべての変数と外部要素の組み合わせについてチェックが入るかを確認する．全て確認し終わったのち，対応関係にある複数の変数に対応する行について，各列でいずれかの行についてにしかチェックが入っていない列に対応する外部要素は，変数からの参照が一致していない外部要素とする．このチェックを全ての対応関係にある変数の組（表 3 の 2 列目にあたる）について調査し，変数からの参照が一致していないと最後まで判定されなかった部分を変数からの参照が一致した外部要素とする．以降，外部要素は変数からの参照が一致していない場合，クローンセット内でその外部要素は差分があるという．

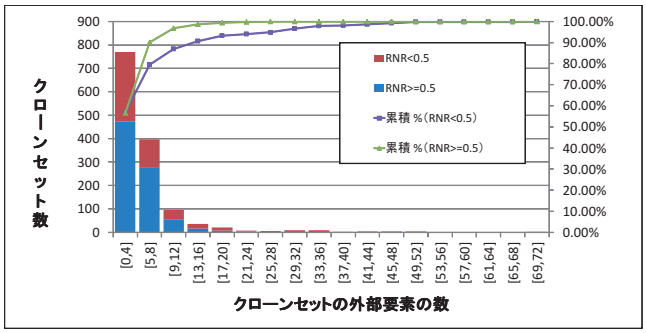

図 3 ArgoUML における外部要素ごとのクローンセットの数

3.4 結果

調査対象となる各プロジェクトのソースコードを解析した結果を以下に示す．

3.4.1 RQ1 : 各コードクローンにおいて，周辺コードはどれだけ多いか

コードクローンにおいて，周辺コードはどれぐらいあるかを調べるために，各クローンセットにおける，外部要素の数と最内制御ブロックの頻度とその種類別の内訳を調査した．

調査対象の各プロジェクトについて外部要素の数ごとにクローンセットの数を数えた結果のうち，ArgoUML, jEdit, muCommander での結果をそれぞれ**図 3**, **図 4**, **図 5** に示す．これらの図では，RNR が 0.5 以上のクローンセットのみを対象とした結果も示している．これは，自動生成されたコードクローンが外部要素の数に影響を与えているか見るためである．

表 3 外部要素一致判定のための表

		(int型) 仮引数1	(int型) 仮引数2	(String型) 仮引数1	(int型) Field1	(double型) Field1	(クラス型) Field3	(int型) getX	(int型) Init
クローンペアの1つ目の変数	クローン1の**x1**	○	×	×	×	×	×	○	×
	クローン2の**x1**	○	×	×	×	×	×	×	○
クローンペアの2つ目の変数	クローン1の**y1**	○	○	×	○	×	○	×	×
	クローン2の**y1**	○	×	○	×	○	×	×	×
			↑ 差分	↑ 差分	↑ 差分	↑ 差分	↑ 差分	↑ 差分	↑ 差分

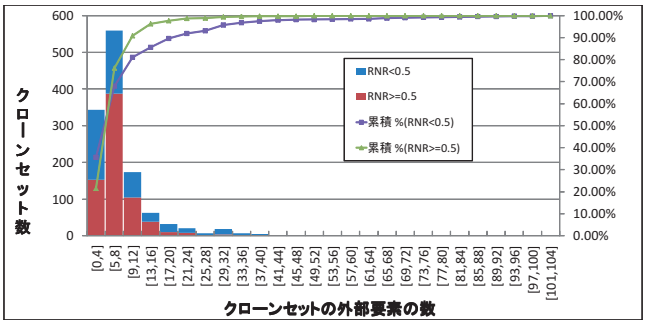

図 4　jEdit における外部要素ごとのクローンセットの数

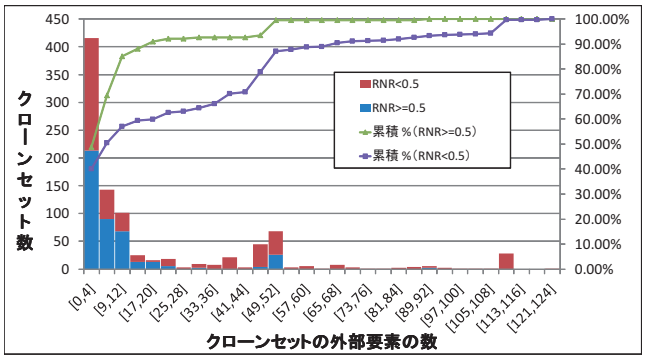

図 5　muCommander における外部要素ごとのクローンセットの数

図 3〜図 5 でわかることとして，各プロジェクトにおいて，外部要素が 12 個以下となるクローンセットは全体の 70% 以上を占めていた．また，この傾向は RNR が 0.5 以上の場合においても同様であった．そのため，自動生成されたかに関係なく，クローンセットは多くの場合 12 個以下の外部要素を持つといえる．

調査対象のプロジェクトにおける各コードクローンの最内ブロックの種類別の頻度を**表 4** に示す．表 4 から，多くの場合，大半の最内ブロックはメソッドとなっている．これは，コードクローンがメソッド内に閉じる制御ブロック内にない場合が多いことを示している．しかし，コードクローンがメソッド内の制御ブロック内にある場合には，if や switch で構成される制御ブロックが最内制御ブロックとなることが多かった．以上の結果から RQ1 への答えは「クローンセットは多くの場合，少数の外部要素を持ち，また，if や switch による最内制御ブロックを持つことがある」となる．

表 4　最内ブロックの種類と頻度

プロジェクト名	メソッド	if	switch	for	while
ArgoUML	3989	1012	225	13	37
DataCrow	4860	527	92	6	55
jEdit	2822	596	97	76	1123
jwebmail	197	37	19	0	2
LaTeXDraw	6745	354	18	15	374
muCommander	2432	710	48	18	178
SweetHome3D	3510	4661	30	1	15

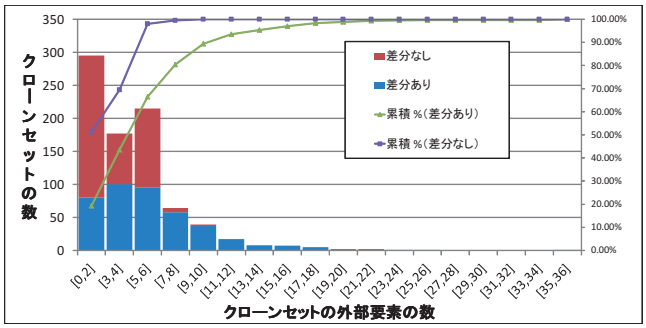

図 6　ArgoUML における外部要素の数に対する RNR が 0.5 以上のクローンセットの個数とその差分の有無による内訳

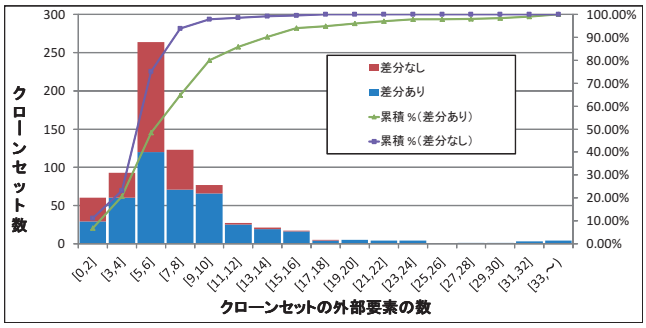

図 7　jEdit における外部要素の数に対する RNR が 0.5 以上のクローンセットの個数とその差分の有無による内訳

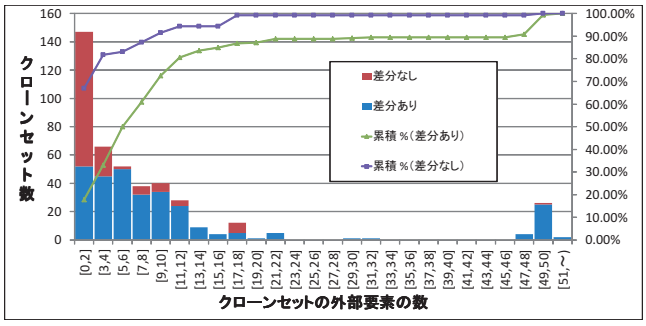

図 8　muCommander における外部要素の数に対する RNR が 0.5 以上のクローンセットの個数とその差分の有無による内訳

3.4.2　RQ2：各コードクローンの周辺コードは，同一のクローンセットの中でどれだけ一致するか

RQ2 に答えるため，調査対象の各プロジェクトについて，RNR が 0.5 以上のクローンセットのうち，外部要素に差分がある場合とない場合がそれぞれいくつあるか，また，差分がある場合変数名に違いがあるかを調べた．

外部要素の数ごとの外部要素に差分のあるクローンセットの数と差分のないクローンセットの数を調べた結果のうち，ArgoUML，jEdit，muCommander についての結果をそれぞれ**図 6**，**図 7**，**図 8** に示す．これらの図から共通して，外部要素が少数であるとき，クローンセットの外部要素に差分がない場合が多くなっている．

外部要素のうち，差分となるものが占める割合ごとのクローンセットの数を，RNR が 0.5 未満，RNR が 0.5 以上かつ変数名が同じでない場合，RNR が 0.5 以上かつ変数名が

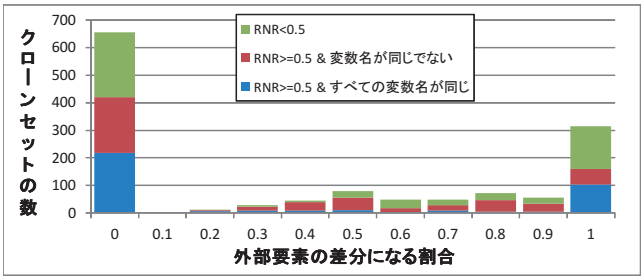

図 9 ArgoUML における差分となる外部要素が外部要素全体に占める割合ごとのクローンセットの個数と，RNR と変数名の一致による内訳

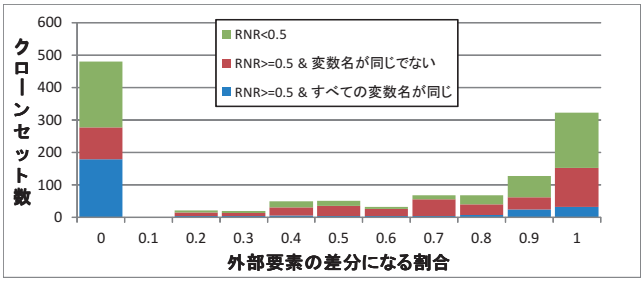

図 10 jEdit における差分となる外部要素が外部要素全体に占める割合ごとのクローンセットの個数と，RNR と変数名の一致による内訳

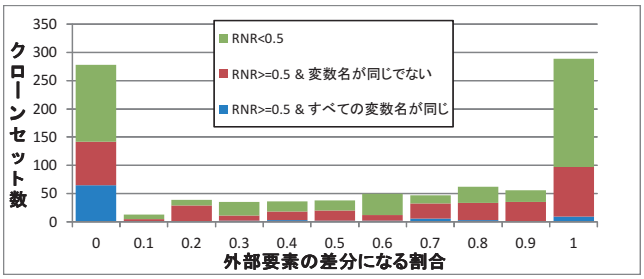

図 11 muCommander における差分となる外部要素が外部要素全体に占める割合ごとのクローンセットの個数と，RNR と変数名の一致による内訳

同じである場合に分けた結果について，ArgoUML，jEdit，muCommander についての結果をそれぞれ**図 9**，**図 10**，**図 11** に示す．これらの図では，共通して，差分が占める割合が 0 の場合と，0.9 より大きく 1 以下である場合にクローンセットが集中している．また，割合が 0.9 より大きい場合であっても，変数が同じである場合や，割合が 0 の場合でも変数名が同じでないという場合があった．

各プロジェクトにおける最内ブロックが異なるクローンセットの数と，全体に占める割合を**表 5** に示す．

表 5 では，最内制御ブロックが異なるクローンセットが 6〜25% 存在したことを示しており，どのプロジェクトにおいても，最内ブロックが異なるクローンセットが含まれているといえる．

以上の結果から，RQ2 に対する答えは「外部要素が一致する場合が多いが，ほとんど異なる場合も同様に多い．また，最内ブロックについては，異なっているクローンセットも存在する」となる．

3.5 考察

RQ1 と RQ2 に対する答えから，多くのクローンセットに外部要素が存在し，それらに差分があることが多いということがわかった．また，最内制御ブロックを持つコードクローンもあり，クローンセット内でそれらの最内制御ブロックが統一されていない事例もあった．このことから，コードクローンは多くの場合，その周辺コードに依存しており，クローンセット内であっても周辺コードが異なるといえる．

また，RQ2 の結果より，外部要素がすべて差分になるクローンセットであったとしても，そのクローンセットに含まれる全てのコードクローンの変数名が同じである場合があった．これらのコードクローンには開発者が見た時，コードの文面が類似しているため，同じ処理を行っていると判断されやすく，誤った理解を引き起こす可能性がある．

以上のことから，コードクローンを正確に理解するために，コードクローンの周辺コードを調査するためのツールが必要ではないかと考える．

3.6 妥当性

本調査では 7 つのオープンソースソフトウェアを実験対象として利用し，コードクローンの周辺コードへの依存性を調査した．オープンソースソフトウェア全体から考えると，使用したソフトウェアの数は小さいため，本調査から得られた結果を一般化するのは難しい．しかしながら，使用したソフトウェアすべてで同様の傾向を確認できており，他のソフトウェアでも類似した傾向を見ることができるのではないかと考える．

4. 関連研究

本研究と同様にコードクローンの周辺コードに着目した研究として Jiang ら [3] の研究がある．Jiang らは，コードクローン同士の周辺コードを比較し，不一致が生じた場合，バグの可能性があると主張している．また，周辺コードの不一致の分類などを定義している．Jiang らは周辺コードとして，コードクローン含むを最内制御ブロックを指定し

表 5 各プロジェクトに対する最内ブロックが異なるクローンセット数

プロジェクト名	全体	異なる	パーセント (%)
ArgoUML	1360	130	9
DataCrow	1137	79	7
jEdit	1242	112	9
jwebmail	113	7	6
LaTeXDraw	967	244	25
muCommander	943	40	4
SweetHome3D	1569	169	11

ている．それに対して，本研究では，周辺コードとして最内制御ブロックに加え，コードクローン内に存在する変数のデータフローに出現する外部要素も指定している．

Jiang らはコードクローンの周辺コードの不一致を利用して，バグがありそうなところを特定し，フィルタリングをかけ，結果を開発者に提供している．それに対し，本研究では，ソフトウェアに存在する各コードクローンの周辺コードを解析し，各周辺コードの違いや，周辺コードからコードクローンへ与える影響を調査している．

Xing ら [20] はコードクローンを検出し，クローンセット内の選択されたコードクローン同士のプログラム依存グラフを比較することでコードクローンとその周りのコードのデータフローと制御フローの差を特定する手法を提案している．それに対し，本研究はコードクローンのデータフローの差として，データフローそのものの差ではなくて，データフローの元となる外部要素を比較している．

5. まとめと今後の課題

本研究では，コードクローンの周辺コードへの依存性を調査した．実験対象として 7 つのオープンソースソフトウェアを用いて調査した結果，コードクローンには周辺コードが存在することが多く，コードクローン間で周辺コードが異なる場合も多く存在していた．また，外部要素がすべて差分になるクローンセットであったとしても，そのクローンセットに含まれる全てのコードクローンの変数名が同じである場合があった．

今後の課題として，コードクローンの周辺コードを調査するためのツールを構築し，実際のソフトウェア開発へ適用して評価することが挙げられる．

謝辞 本研究は科研費（課題番号:23680001，24700029）の助成を受けたものである．

参考文献

[1] 肥後芳樹，楠本真二，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，Vol. J91-D, No. 6, pp. 1465–1481 (2008).
[2] Kapser, C. J. and Godfrey, M. W.: “Cloning considered harmful” considered harmful: patterns of cloning in software, *Empirical Software Engineering*, Vol. 13, No. 6, pp. 645–692 (2008).
[3] Jiang, L., Su, Z. and Chiu, E.: Context-based detection of clone-related bugs, *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*.
[4] Krinke, J.: Is cloned code older than non-cloned code?, *IWSC ’11 Proceedings of the 5th International Workshop on Software Clones*.
[5] Harder, J. and Gode, N.: Cloned code: stable code, *Journal of Software: Evolution and Process* (2012).
[6] Bellon, S., Koschke, R., Antoniola, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transaction Software Engineering*, Vol. 33, No. 9, pp. 577–591 (2007).
[7] Baxter, I., Yahin, A., Moura, L., Anna, M. S. and Bier, L.: Clone detection using abstract syntax trees, *Proceedings of International Conference on Software Maintenance ’98*, pp. 368–377 (1998).
[8] Roy, C. and Cordy, J.: A survey on software clone detection research, Technical report, Queen’s University at Kingston (2007).
[9] Koschke, R.: Survey of research on software clones, *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings (2007).
[10] Jarzabek, S. and Li, S.: Eliminating redundancies with a ” composition with adaptation” meta-programming technique, *European Software Engineering Conference and Foundations of Software Engineering*, pp. 96–105 (2003).
[11] Rajapakse, D. and Jarzabek, S.: Using server pages to unify clones in web applications: A trade-off analysis, *International Conference on Software Engineering*, pp. 116–126 (2007).
[12] Prechelt, L., Malpohl, G. and Philippsen., M.: Finding plagiarisms among a set of programs with JPlag, *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016–1038 (2002).
[13] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code, *IEEE Transaction Software Engineering*, Vol. 32, No. 3, pp. 176–192 (2006).
[14] Basit, H. and Jarzabek, S.: Detecting higher-level similarity patterns in programs, *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 156–165 (2005).
[15] Mayrand, J., Leblanc, C. and Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics, *Proceedings of International Conference on Software Maintenance ’96*, pp. 244–253 (1996).
[16] Kontogiannisa, K., DeMori, R., Merlo, E., Galler, M. and M.Bernstein: Experiment on the automatic detection of function clones in a software system using metrics, *Automated Software Engineering*, Vol. 3, pp. 77–108 (1996).
[17] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Transaction Software Engineering*, Vol. 28, No. 1, pp. 654–670 (2002).
[18] Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Method and Implementation for Investigating Code Clones in a Soft-ware System, *Information and Software Technology*, Vol. 49, No. 9-10, pp. 985–998 (2007).
[19] Ishio, T., Etsuda, S. and Inoue, K.: A Lightweight Visualization of Interprocedural Data-Flow Paths for Source Code Reading, *Proceedings of the 20th International Conference on Program Comprehension*, pp. 37–46 (2012).
[20] Xing, Z., Xue, Y. and Jarzabek, S.: CloneDifferentiator: Analyzing clones by differentiation, *ASE ’11 Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 576–579 (2011).