特別研究報告

題目

コードクローンの理解支援を目的とした
コードクローン周辺コードの解析

指導教員

井上 克郎 教授

報告者

Buyannemekh Odkhuu

平成 25 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 24 年度 特別研究報告

コードクローンの理解支援を目的としたコードクローン周辺コードの解析

Buyannemekh Odkhuu

内容梗概

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，ソースコード中に存在する，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．もし，コードクローンを持つコード片に欠陥が含まれている場合，そのすべてのコードクローンにも欠陥が含まれている可能性が高く，それらを修正するのは大きなコストとなる．一方，すべてのコードクローンが有害であるわけではないと言われている．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．各コードクローンの動作は，その周辺のコードに依存する．しかしながら，実際にどの程度コードクローンがその周辺のコードに依存しているかはわかっていない．

そこで，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，違い，それらを従来のコードクローンに関するメトリクスで説明できるかについて調査した．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した．また，周辺コードの量や，違いは従来のコードクローンメトリクスでは説明が難しいこともわかった．また，調査結果に基づき，コードクローンを調査するためのツールを試作した．本ツールは周辺コードについての情報を量の表示と，ソースコード上での視覚化によって表現している．

主な用語

コードクローン
ソフトウェア保守
周辺コード

目 次

1　まえがき

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存在する．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．

本研究ではコードクローンを正確に調査するため，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．しかしながら，実際にどの程度コードクローンがその周辺のコードに依存しているかはわかっていない．

そこで，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，違い，それらを従来のコードクローンに関するメトリクスで説明できるかについて調査した．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した．また，従来のコードクローンメトリクスでは説明が難しいこともわかった．また，調査結果に基づき，コードクローンを調査するためのツールを試作した．本ツールは周辺コードについての情報を量の表示と，ソースコード上での視覚化によって表現している．

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，その方法と結果について述べる．４章では，試作したコードクローン周辺コード調査ツールを紹介する．５章では，関連研究を，6 章ではまとめと今後の課題について述べる．

2　背景

本章では，本研究の背景として，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．

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．

一般的に，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　定義と分類

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する方法が存在する．そのどれもが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．

2.1.2　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．

2.1.3 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統一されない．

2. パフォーマンス関連
   コードクローンを統一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. ソフトウェア開発時の実行
   一部の実験的なコードは統合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い．よって，コードクローンが常に存在し，バグ原因になる可能性がある．しかしながら，すべてのコードクローンが有害ではない．

2.1.4 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．

コードクローン検出システムの一例として CCFinder[17] を説明する．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．

2.1.5 メトリクス

コードクローンのメトリクスとは，コードクローンの性質，特徴を表す値のことである．ソフトウェアにはコードクローンになるコード片が幅広く存在し，それらのコード片から重要なコードクローンを手作業で探し出し，管理するのは困難である．そこで，メトリクスは重要なコードクローンを特定するのに役に立つ [17]．以下に，コードクローンのメトリクスの例を示す．

- LEN：コードクローンの平均長さ（トークン数）を示すメトリクス [17]

- NIF：コードクローンが存在するファイルの数を示すメトリクス
- POP：コードクローンとなるコード片の数を示すメトリクス [17]
- RAD：コードクローンとなるコード片が存在するファイルがファイルシステムの中でディレクトリ構造的にどれだけ分散しているかを示すメトリクス [17]
- RNR：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 [18]

2.1.6 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 [3] 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 は関数呼び出しで値を取得する場合，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 と 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．

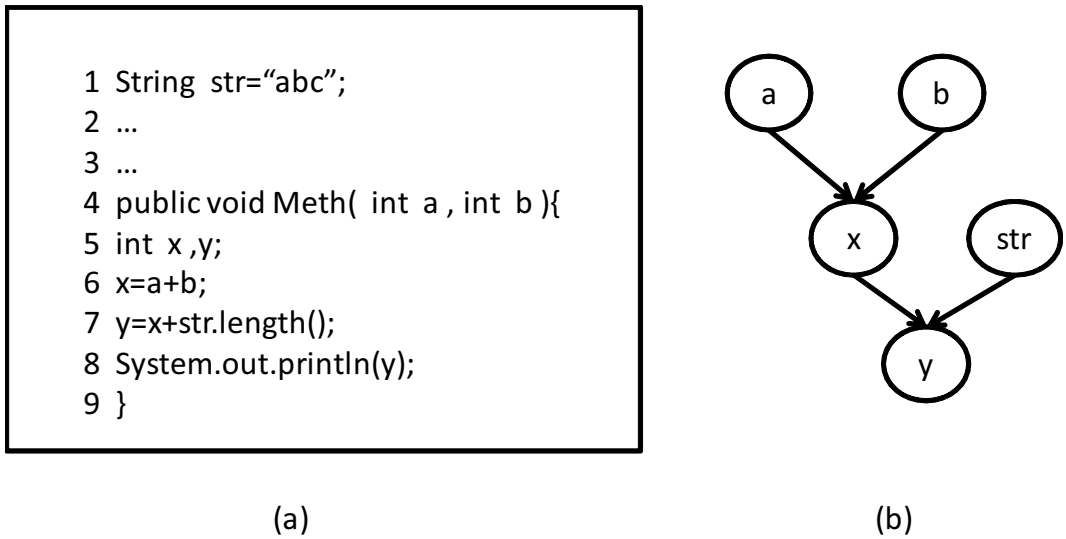


図 1: データフロー図の例，(a) ソースコード，(b)(a) の変数 y に対するデータフロー

図 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．

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．

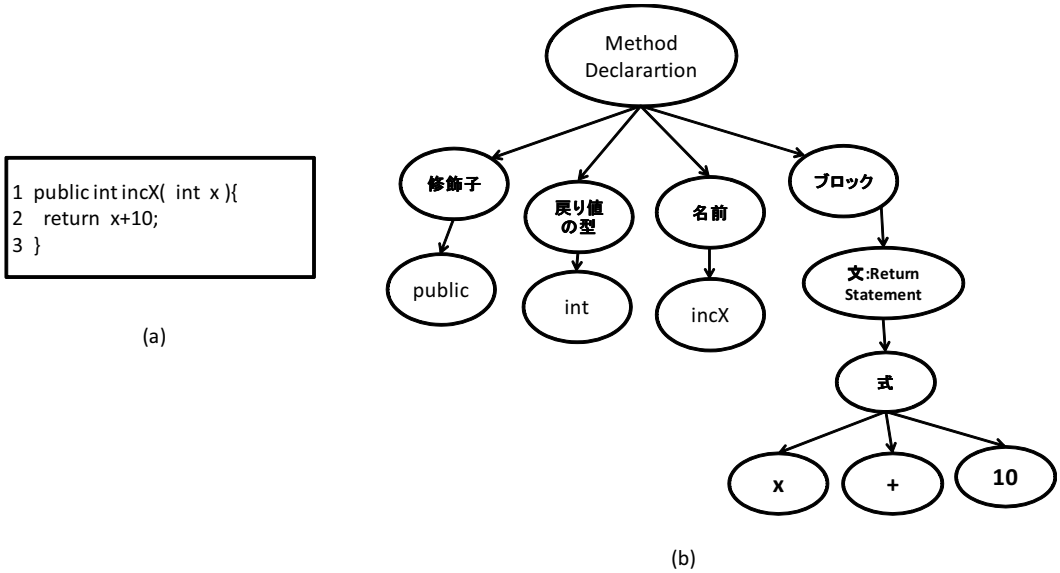


図 2: Eclipse の JDT によって作成される AST のイメージ図

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，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

RQ3：周辺コードに基づくメトリクスはコードクローンメトリクスと同様な傾向を持つか

3.3　調査方法

初めに CCFinder を用いてコードクローンとクローンセットを検出する．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，抽出した各コードクローンについて，周辺コードを特定する．

次に，特定した周辺コードに基づき，各リサーチクエスチョンに答えるために必要なデータを取得する．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

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（ただし，1 つのクローンセット内の各コードクローンに出現する変数名が同一かどうかでクローンセットを 2 グループに分類して集計する）

RQ3 RQ1 と RQ2 で用いた数値と既存のコードクローンメトリクス LEN，NIF，POP，RAD，RNR との相関係数

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とファイル数，表 2 にはコードクローンに関する情報を示す．コードクローンに関連する情報とは本研究の対象となるクローンセット数，コードクローン数と変数を含まないコードクローン数と 1 つのメソッドに完全に閉じていないコードクローン数である．

以降，周辺コードの取得方法と，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．

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	ソースコードの行数	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	実験対象 クローンセット数	実験対象 コードクローン数	変数を含まない コードクローン数	メソッドに閉じていない コードクローン数
Argouml	2073	5276	713	514
DataCrow	1337	5540	259	178
LaTeXDraw	1358	7506	391	704
SweetHome3D	2017	8217	448	318
jwebmail	125	255	14	7
jEdit	1740	4714	498	76
muCommander	1496	3386	553	126

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，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上に出現する外部要素を特定する処理の流れを図 3 に示す.

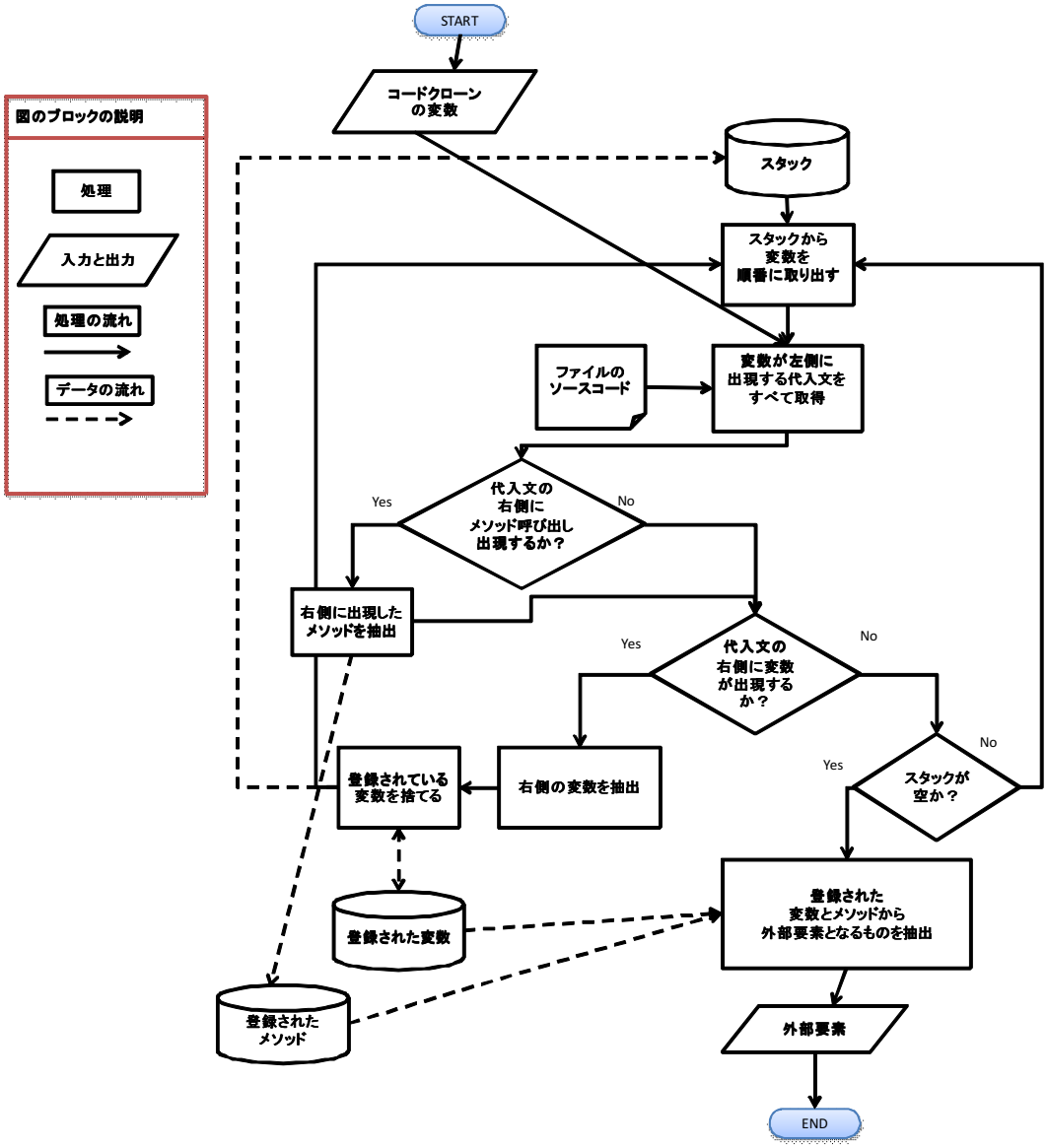


図 3: データフロー上に出現する外部要素を取得する処理の流れ

1. 入力されたコードクローンの変数が左側に出現している代入文を（ソースファイルから）すべて特定する．

2. 特定した代入文の右側にメソッド呼び出しが出現するかをチェックし，出現すれば次のステップに行き，出現しなければ，ステップ 4 へ行く．

3. 右側に出現するメソッド呼び出しからメソッドを抽出して “登録されたメソッド ” に格納する．次のステップへ行く．

4. 特定した代入文の右側に変数が出現するかをチェックし，出現すれば次のステップに行き，出現しなければステップ 10 へ行く．

5. 特定した代入文の右側に出現する変数をすべて抽出する．

6. 抽出した変数が登録された変数の中に含まれるかをチェックし，抽出した変数が既に登録されている場合は抽出した変数を捨て，登録されていない場合は抽出した変数をスタックと “登録された変数 ” に格納する．

7. スタックから変数を順番に取り出す.

8. スタックから取り出した変数が左側に出現している代入文を（ソースファイルから）すべて特定する．

9. 2〜8 を繰り返す (スタックが空になるまで).

10. スタックが空かをチェックし，空であれば “登録された変数 ”，“登録されたメソッド ” から外部要素となるものを抽出し，出力して終了する．空でなかったらステップ 7 へ行く．

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．

3.3.5 コードクローンに対するメトリクス値の計算

コードクローンに対するメトリクスとして LEN，NIF，POP，RAD，RNR の 5 つのメトリクスを工藤の分析ツール [20] を用いて計算する．

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**

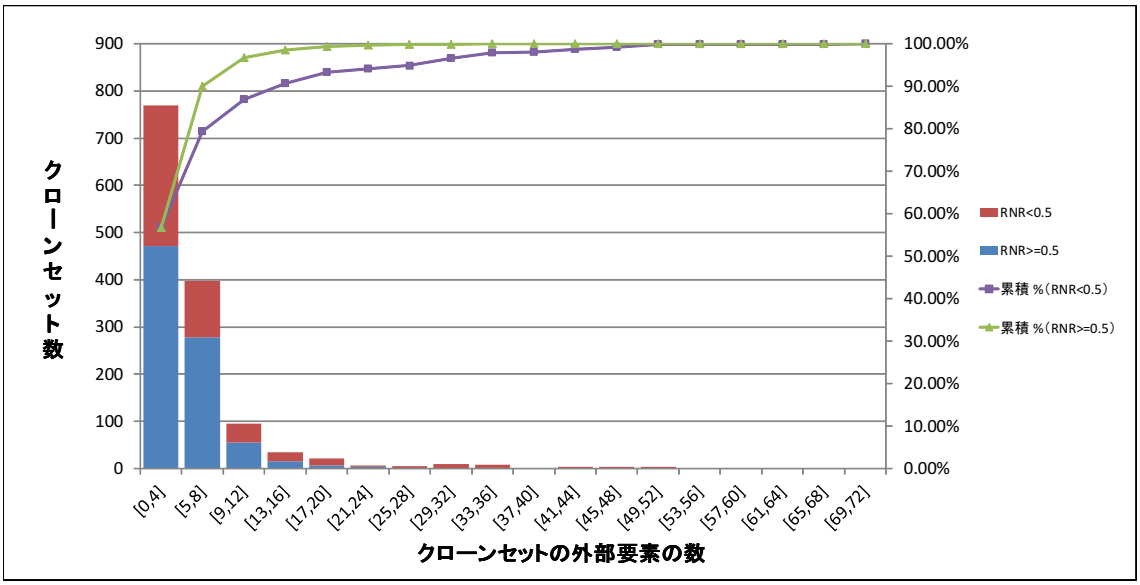


図 4: Argouml のクローンセットに対する外部要素のヒストグラム

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，コードクローンの周辺コードの中にある外部要素数（クローンセット単位で外部要素数を示す）とコードクローンを含む最内制御ブロック種類，頻度を調べた.

図 4 から図 10 に調査対象の各プロジェクトのクローンセットに対する外部要素のヒストグラムを示す.

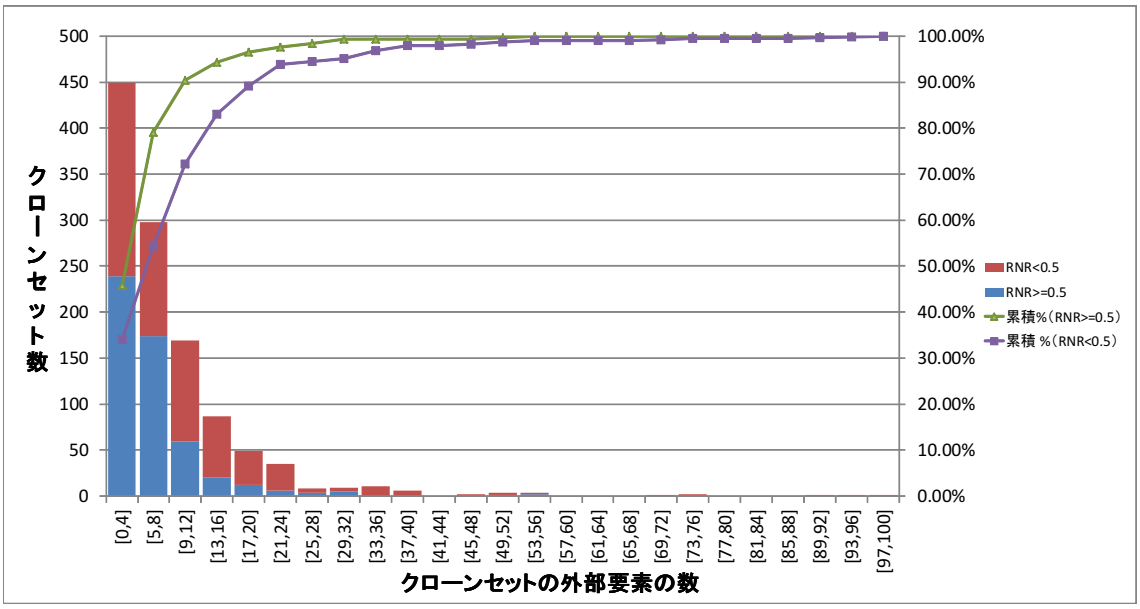


図 5: datacrow のクローンセットに対する外部要素のヒストグラム

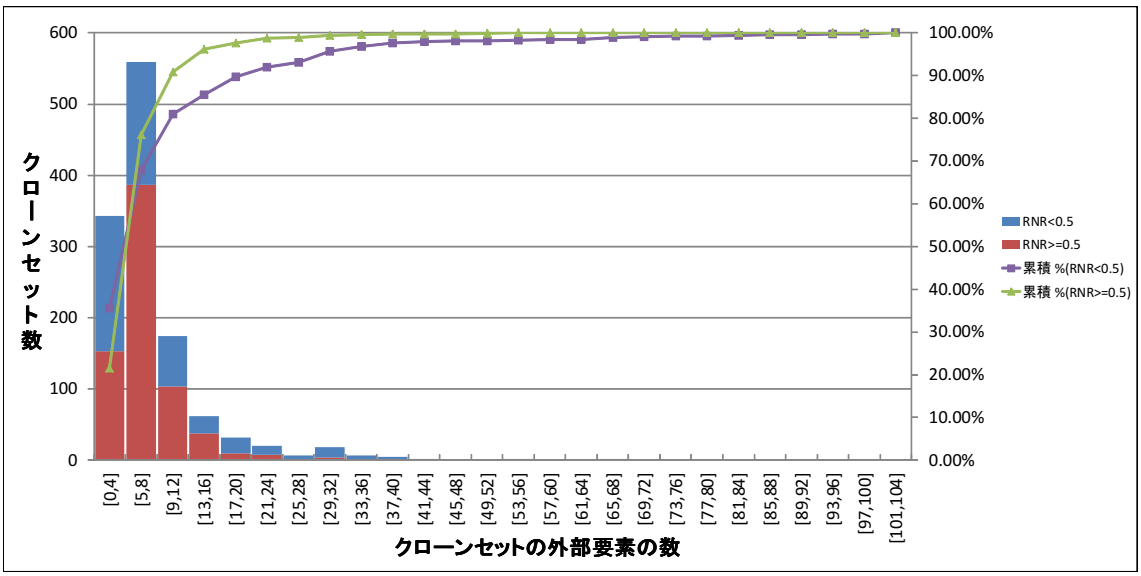


図 6: jEdit のクローンセットに対する外部要素のヒストグラム

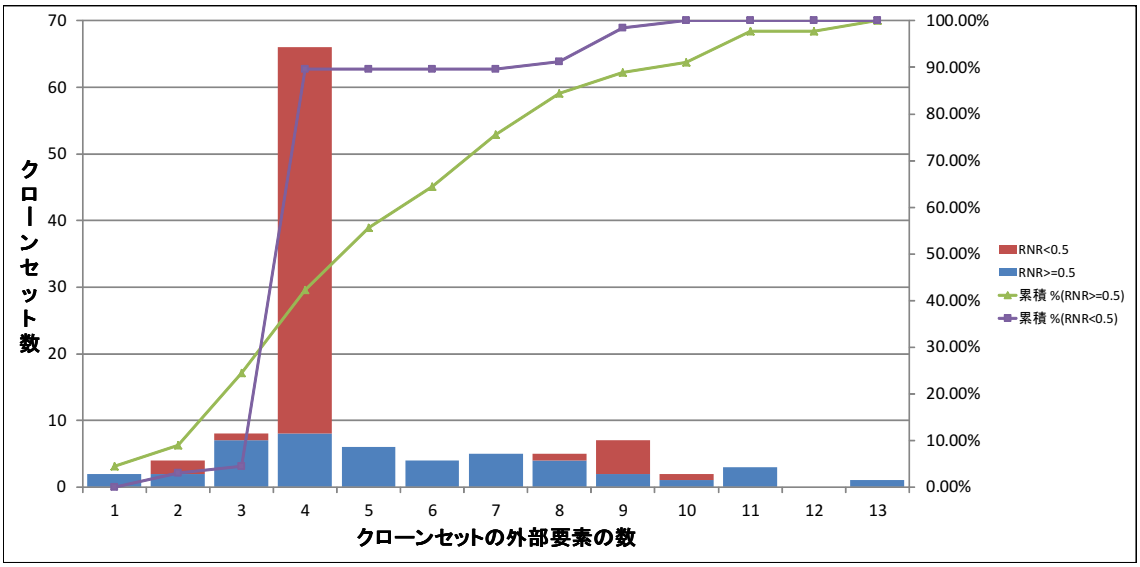


図 7: jwebmail のクローンセットに対する外部要素のヒストグラム

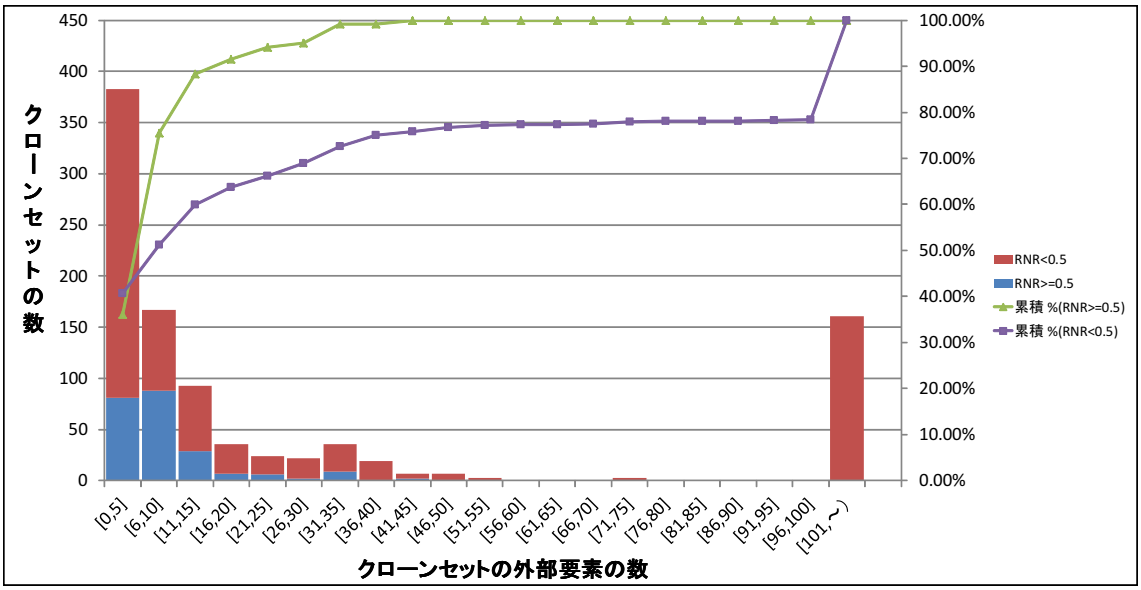


図 8: LaTeXDraw のクローンセットに対する外部要素のヒストグラム

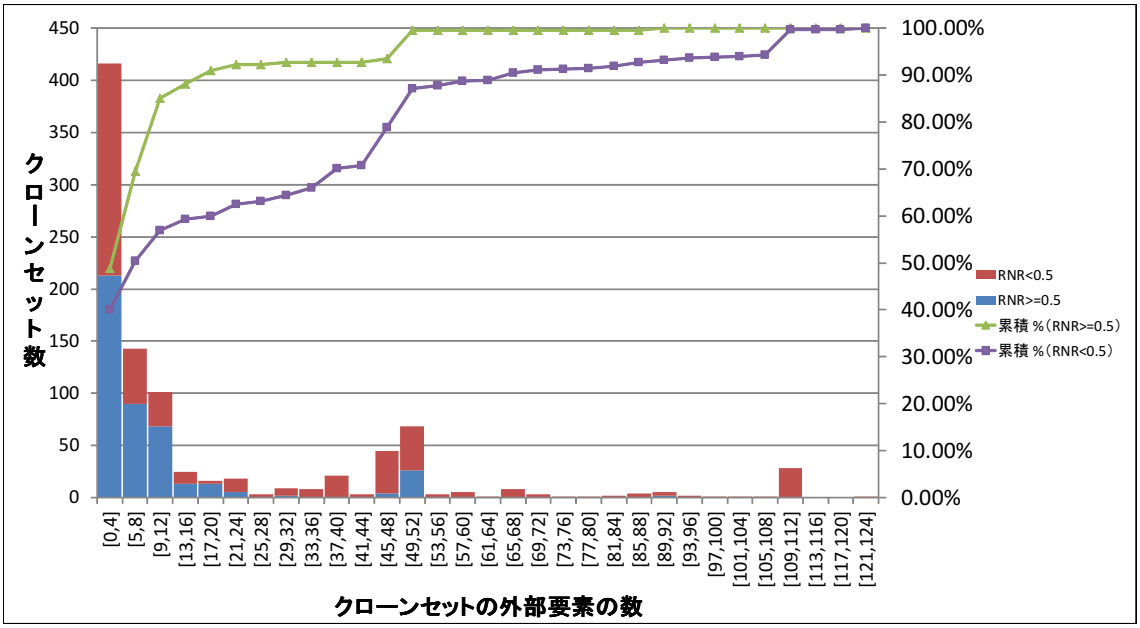


図 9: muCommander のクローンセットに対する外部要素のヒストグラム

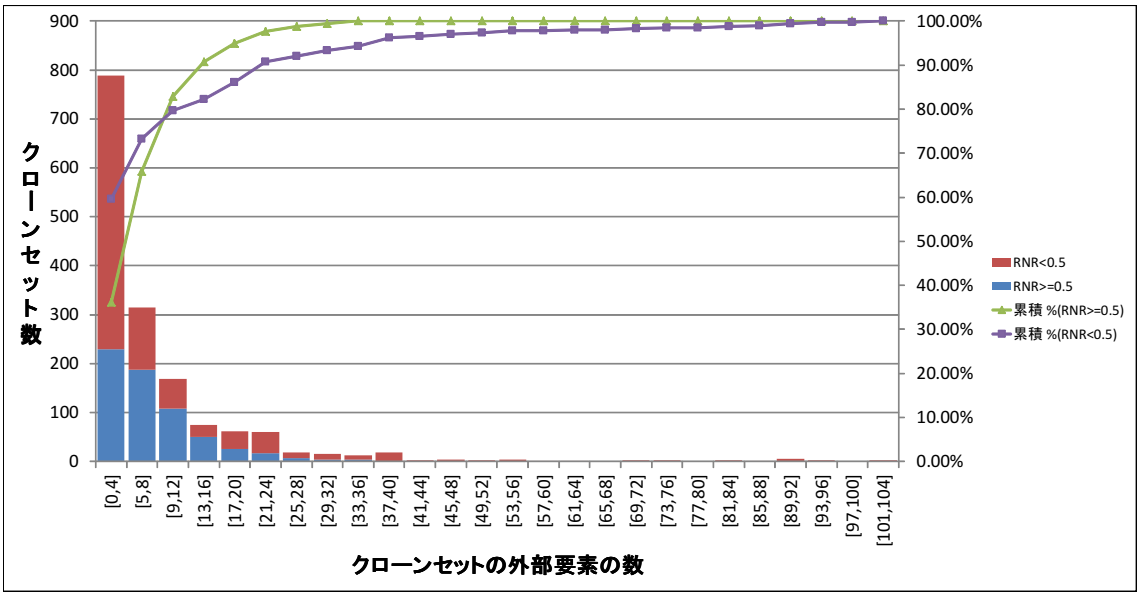


図 10: SweetHome3D のクローンセットに対する外部要素のヒストグラム

図 4〜図 10 からわかるようにプロジェクトの 70〜80 パーセントのクローンセットは外部要素数が 10 個以下であった．RNR≥0.5 のクローンセットにおいても同様なことが言える．つまり，プロジェクトが外部要素を多く持つクローンセットをが少ないと言える．

また，以下の表 4 に各プロジェクトのコードクローンの最内制御ブロックになった制御ブロックの種類とその頻度を示す．

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

表 4 から if，switch 文の方が繰り返し文 while，for などより多くなっている傾向がわかる．また，メソッドと if 文以外の，コードクローンの最内ブロックは少ない．

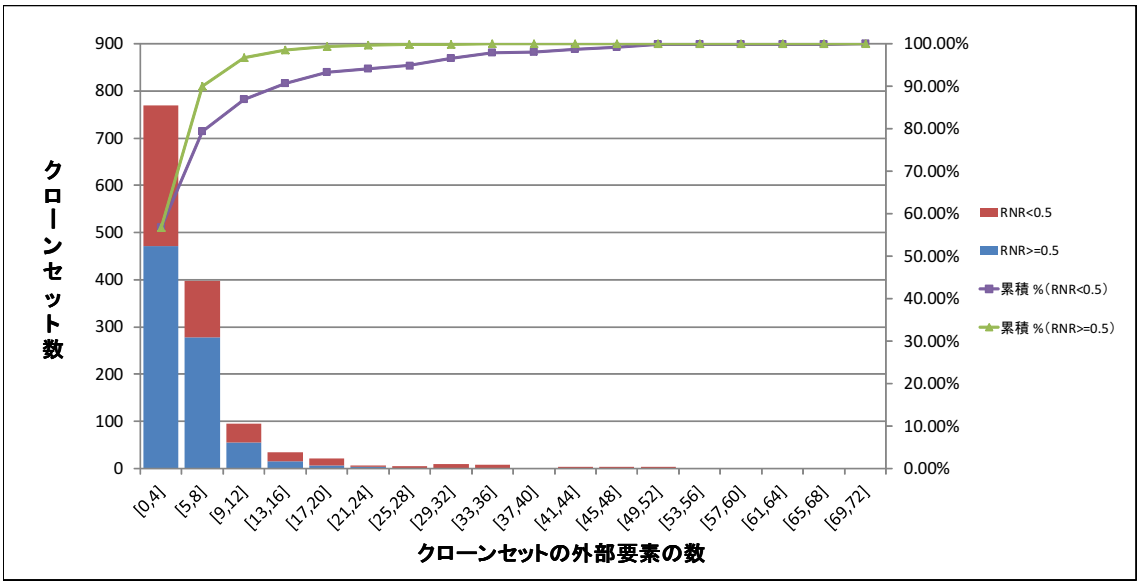


図 11: Argouml の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

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

同一のクローンセットの中でコードクローンの周辺コードはどれだけ一致するかを調べるために，各プロジェクトの外部要素に差分があるクローンセットの外部要素数と外部要素に差分がないクローンセットの外部要素数を求めた．また，クローンセットに対する，外部要素がどれぐらい割合で差分になっているかを調べた．その中で RNR≥0.5 のクローンセットに対して，クローンセット内のコードクローン同士の対応する変数名が同一になっているクローンセット数とそうでないクローンセット数をそれぞれ調査した．

図 11 から図 17 に実験対象プロジェクトの RNR≥0.5 であるクローンセットに対する外部要素のヒストグラムを示す．その際，外部要素は差分があるクローンセットと外部要素は差分がないクローンセットを色で分ける．

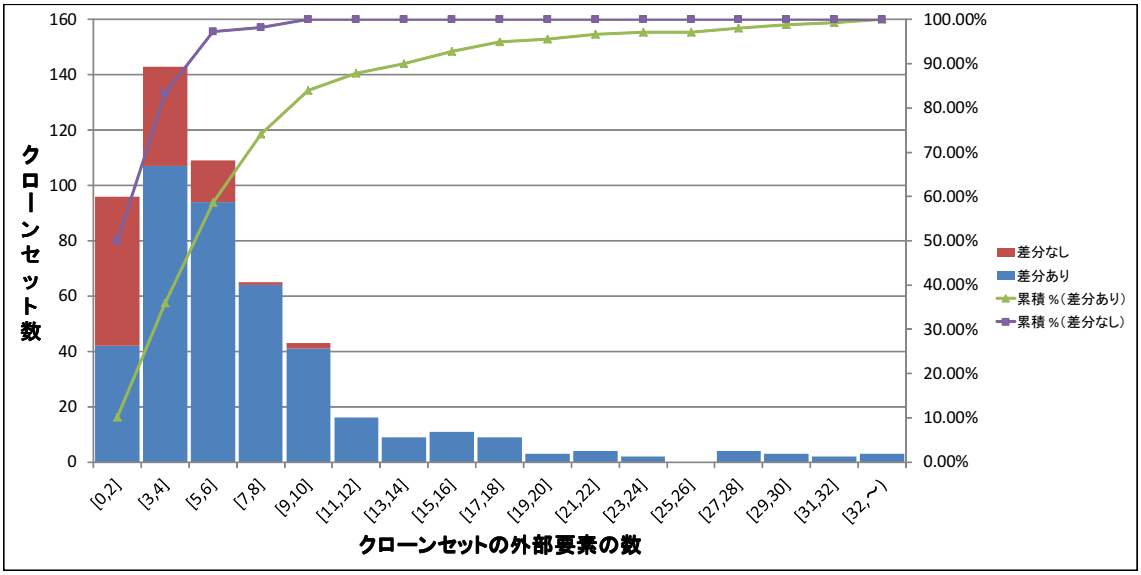


図 12: DataCrow の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

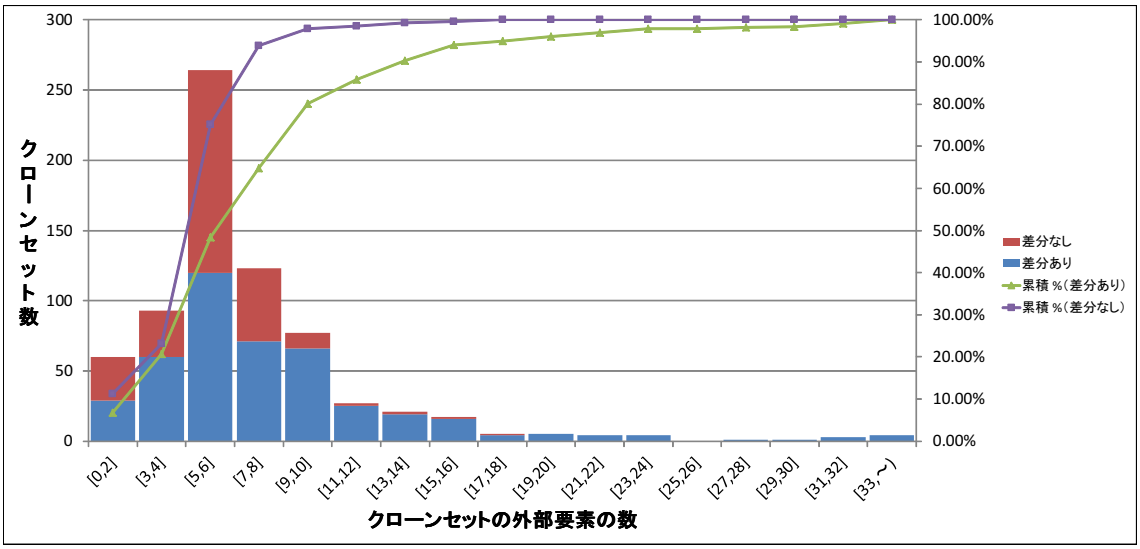


図 13: jEdit の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

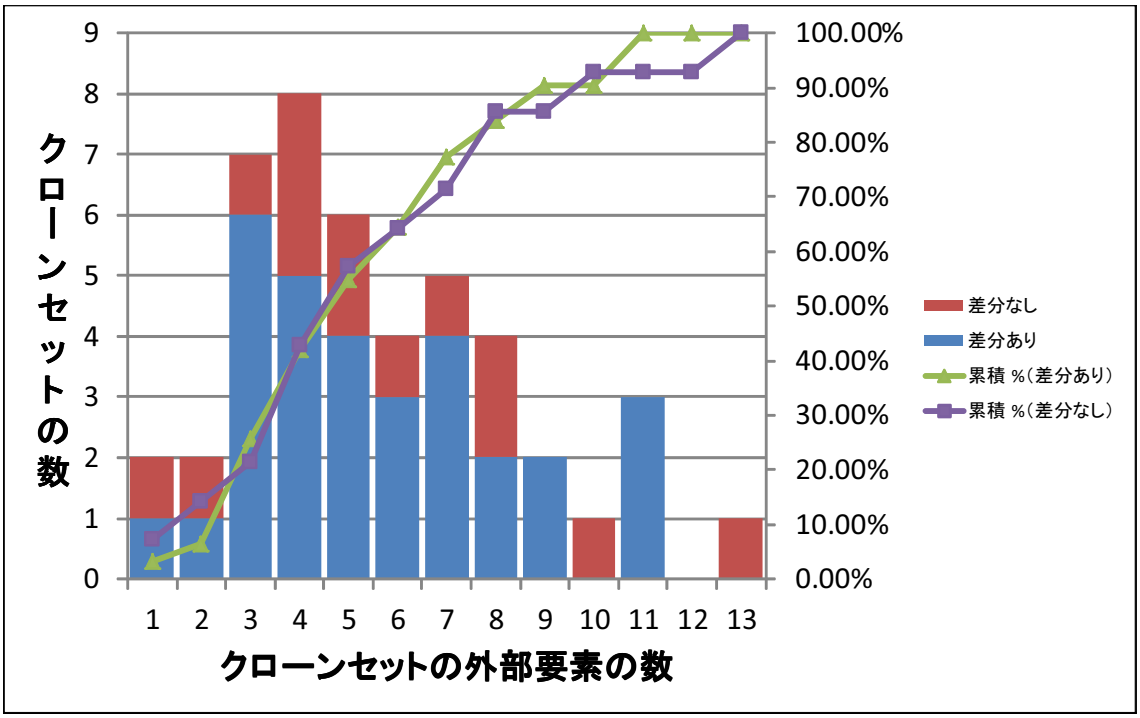


図 14: jwebmail の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

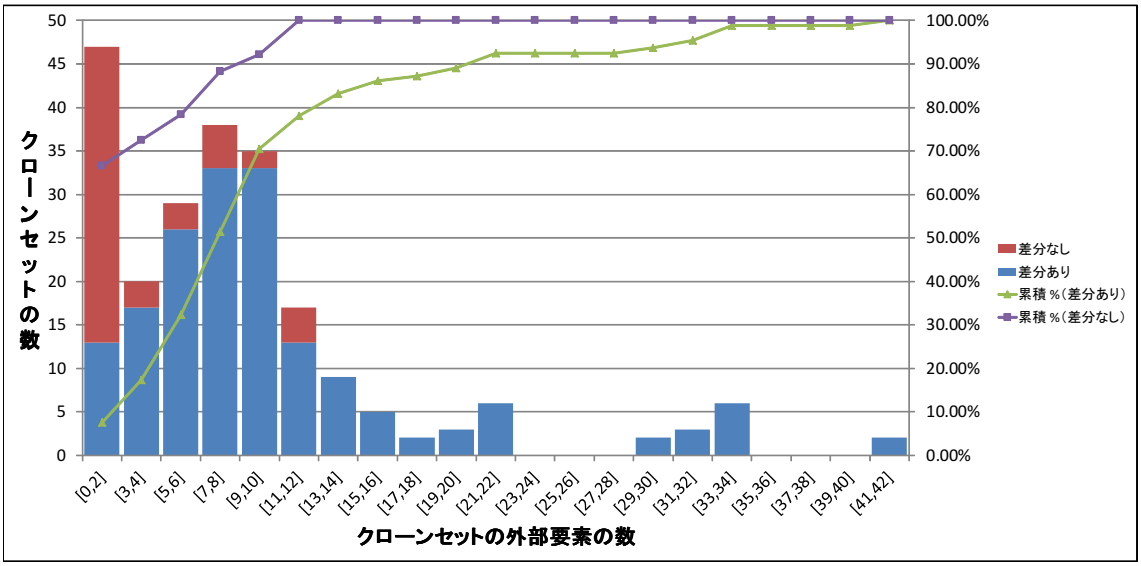


図 15: LaTeXDraw の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

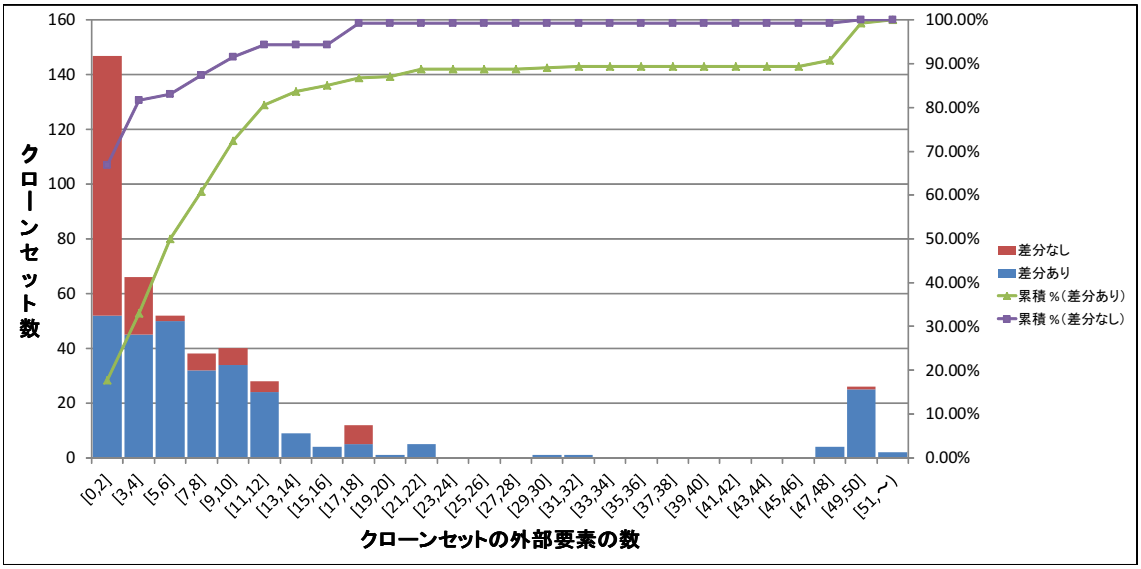


図 16: muCommander の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

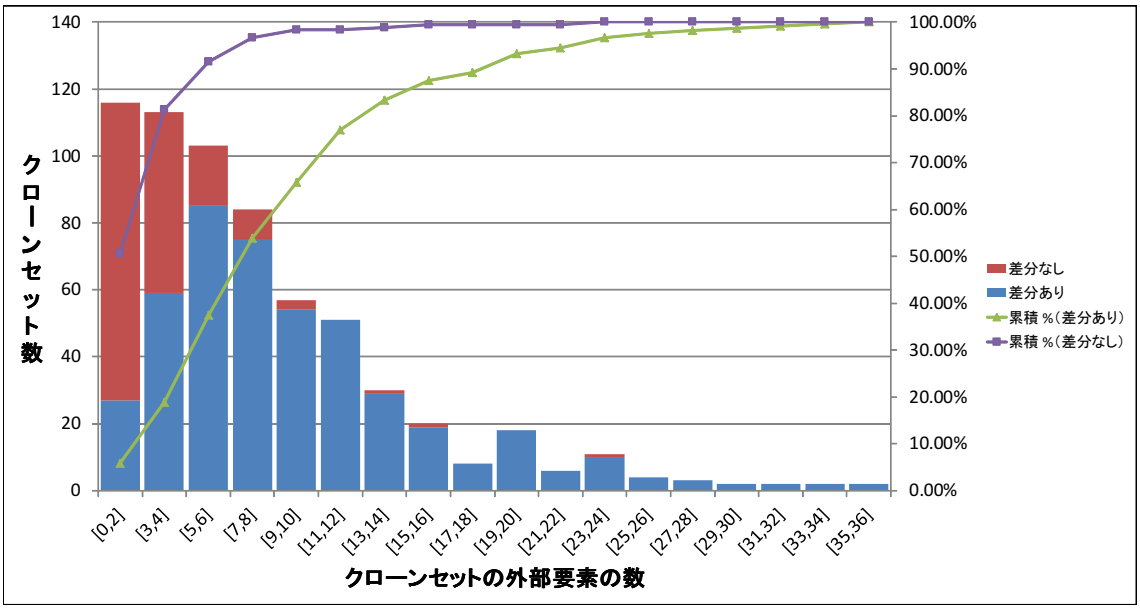


図 17: SweetHome3D の（RNR≥0.5）クローンセットに対する外部要素のヒストグラム

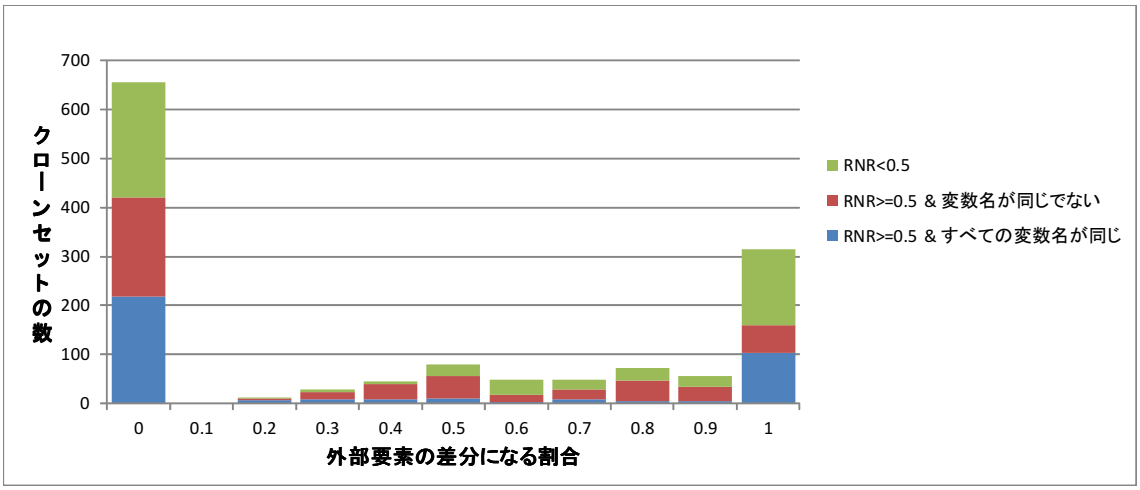


図 18: Argouml のクローンセットに対する外部要素の差分になる割合のヒストグラム

図 11～図 17 から外部要素の数に関係なく外部要素の差分があるコードクローンは存在することをわかる．外部要素が少ないとき，外部要素の差分がないクローンセットが多い．外部要素が多くなるにつれ外部要素の差分があるコードクローンが多い．

図 18 から図 24 に調査対象プロジェクトのクローンセットに対する外部要素の差分になる割合のヒストグラムを示す．本研究では，外部要素の差分になる割合をすべてのクローンセットとコードクローン同士の類似が高い（RNR≥0.5）クローンセットに対して調べた．また，クローンセットとコードクローン同士の類似が高い（RNR≥0.5）クローンセットに対して，クローンセット内のコードクローン同士の対応する変数名が同一で，元となる値が異なるものがどれぐらいか，変数名が異なるが元となる値がどれぐらいかを調べた．

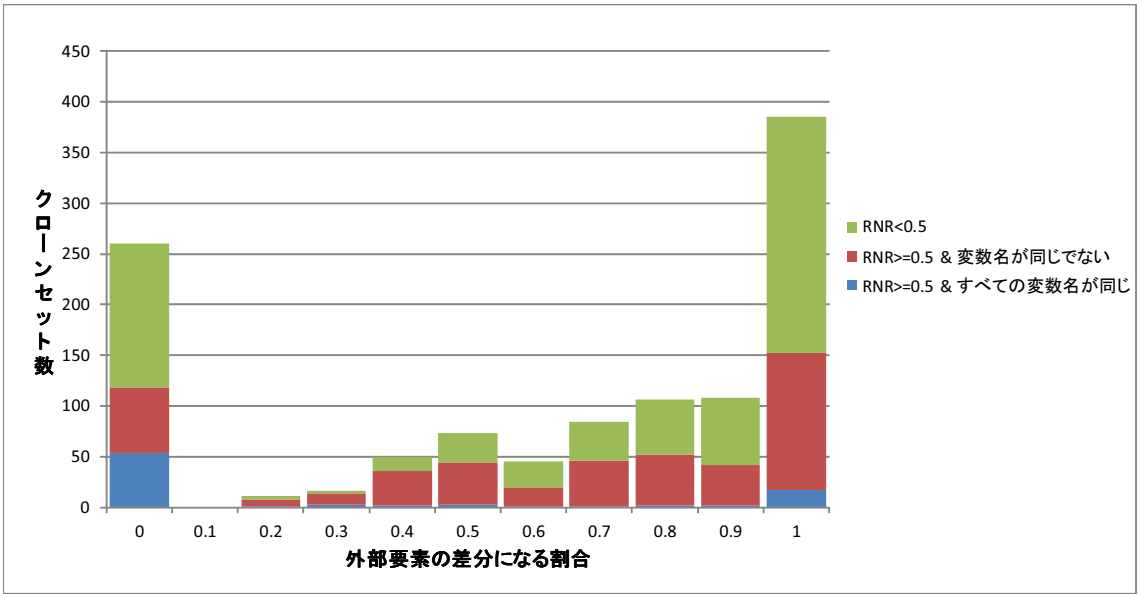


図 19: DataCrow のクローンセットに対する外部要素の差分になる割合のヒストグラム

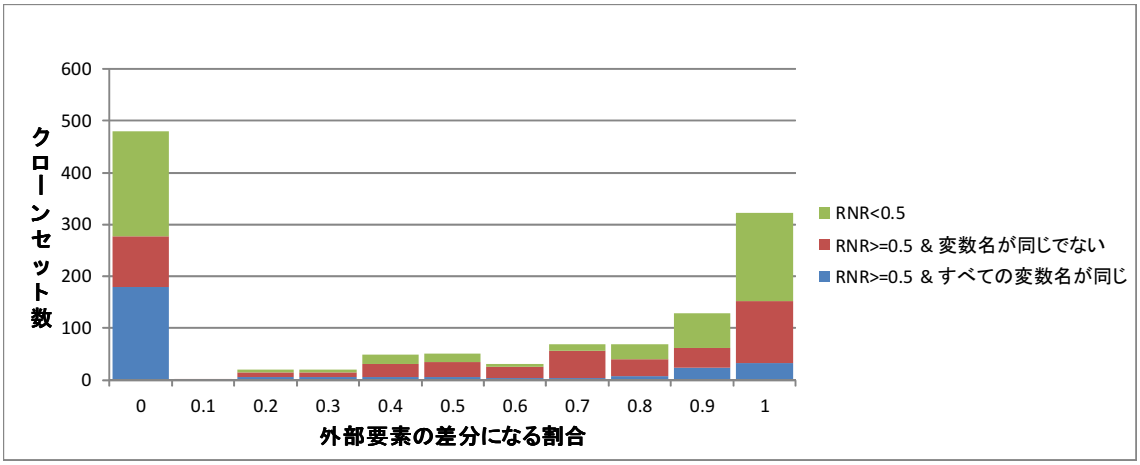


図 20: jEdit のクローンセットに対する外部要素の差分になる割合のヒストグラム

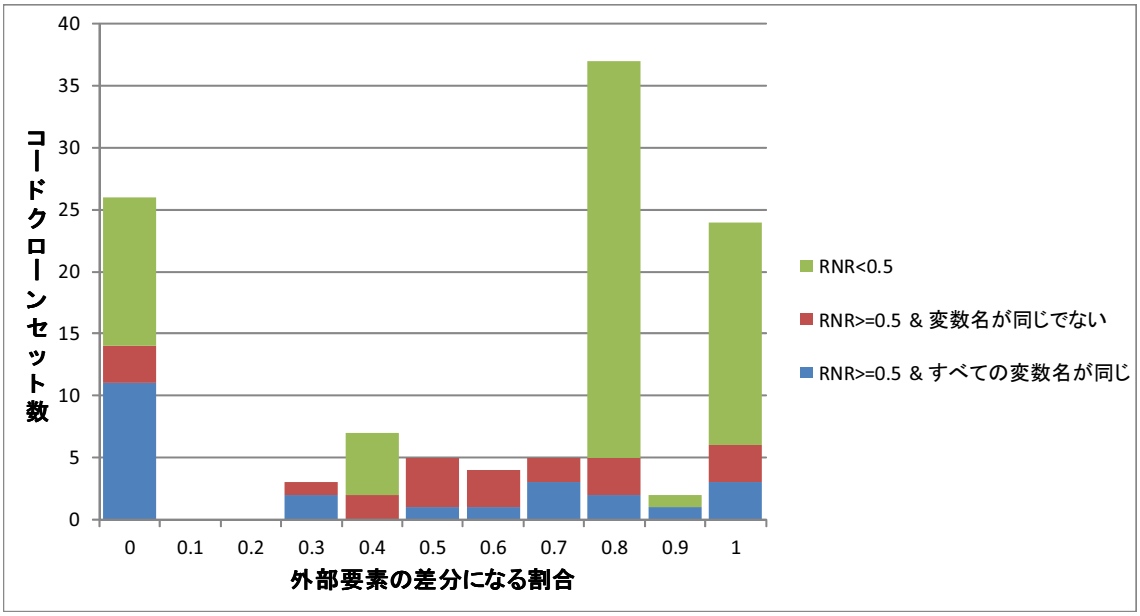


図 21: jwebmail のクローンセットに対する外部要素の差分になる割合のヒストグラム

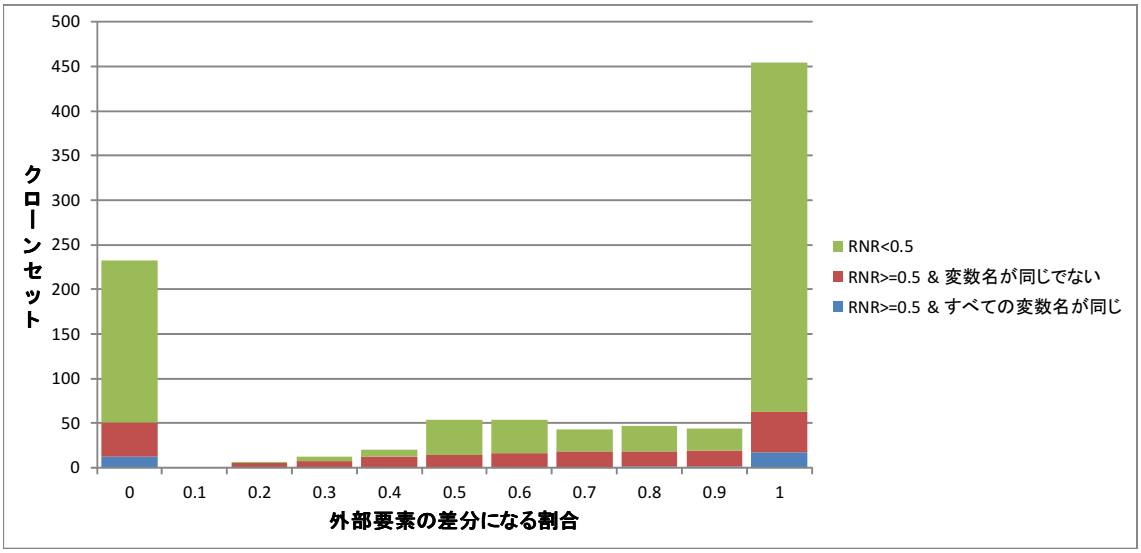


図 22: LaTeXDraw のクローンセットに対する外部要素の差分になる割合のヒストグラム

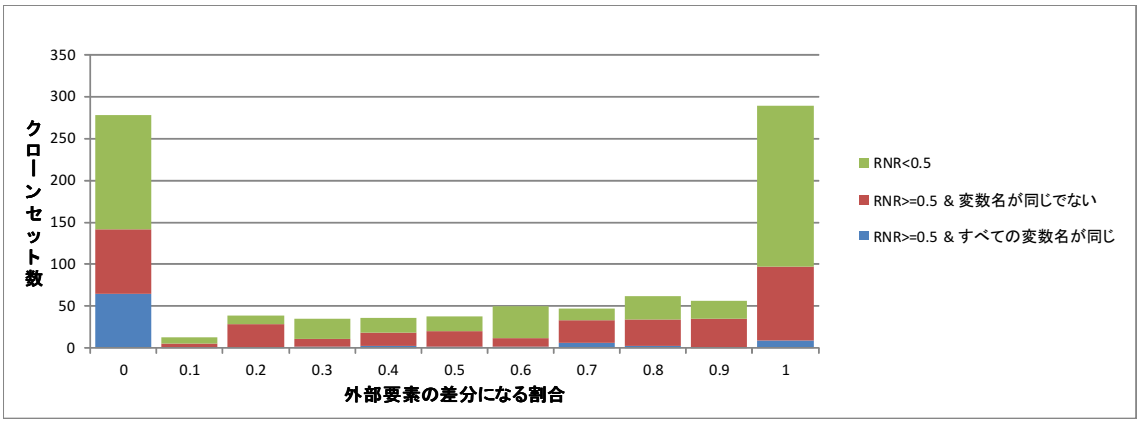


図 23: muCommander のクローンセットに対する外部要素の差分になる割合のヒストグラム

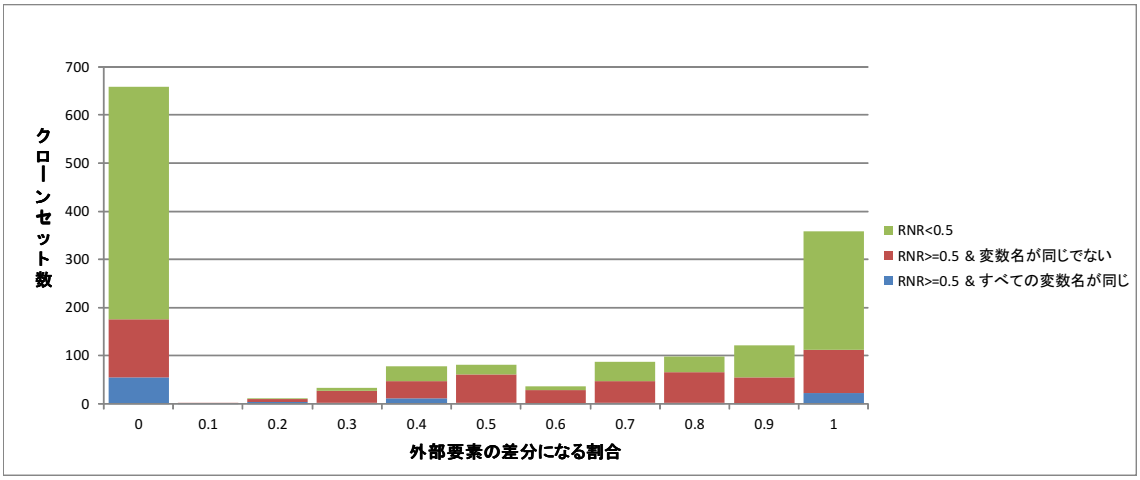


図 24: SweetHome3D のクローンセットに対する外部要素の差分になる割合のヒストグラム

図 18〜図 24 からわかるように各プロジェクトのクローンセットの外部要素がすべて差分になるか，すべて差分にならないかで大きく分けられるのをわかる．つまり各プロジェクトにとって外部要素がすべて差分になるクローンセットが多いことをわかる．また，図 18〜図 24 からコードクローン同士の対応関係のある変数の変数名が同じで，外部要素（元となる値）が異なるクローンセットが RNR が 0.5 以上のクローンセット中にわすかでも（プロジェクトによって多くなる場合もある）存在しているのがわかる．

表 5 に各プロジェクトに対する，最内制御ブロックが異なるクローンセット数を示す．

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

最内制御ブロックが異なるクローンセットは 6〜25 パーセント存在した．

3.4.3 RQ3 : 周辺コードに基づくメトリクスはコードクローンメトリクスと同様な傾向を持つか

周辺コード（外部要素）に基づくメトリクスが既存のコードクローンメトリクスと同様な傾向を持つかを調べるために，外部要素に基づくメトリクスとコードクローンメトリクス間の相関係数を求めた．その結果を表 6 に示す.（表 6 の DiffCloneSet メトリクスは外部要素の差分になる割合を示す）

表 6: 周辺コードのメトリクスとコードクローンメトリクスの相関系

プロジェクト名		LEN	NIF	POP	RAD	RNR
Argouml	外部要素数	0.02897	0.21512	0.41574	0.15976	-0.22210
	DiffCloneSet	0.05426	0.15753	0.25525	0.15021	-0.25526
DataCrow	外部要素数	-0.12648	0.46625	**0.59229**	0.44494	-0.25226
	DiffCloneSet	-0.12852	0.38887	**0.51218**	0.48852	-0.29958
jEdit	外部要素数	-0.15885	**0.64023**	**0.55103**	0.24881	-0.12777
	DiffCloneSet	-0.10919	**0.57934**	0.49340	0.20050	-0.19651
jwebmail	外部要素数	-0.31691	0.35938	0.44236	0.20902	0.32948
	DiffCloneSet	0.34294	0.47447	0.21405	0.3612	-0.14760
LaTexDraw	外部要素数	**0.63307**	0.30114	**0.59376**	0.43252	-0.38277
	DiffCloneSet	**0.65887**	0.25872	**0.57000**	0.40845	-0.38916
muCommander	外部要素数	**0.57332**	0.13673	0.19218	0.11750	-0.41437
	DiffCloneSet	**0.57226**	0.14220	0.15490	0.12041	-0.41428
SweetHome3D	外部要素数	-0.10620	**0.52267**	0.12877	0.12573	-0.05653
	DiffCloneSet	-0.07171	**0.54847**	0.13823	0.10548	-0.10314

表 6 には，どのプロジェクトでも相関係数が高いものはないので，周辺コードの差分に基づくメトリクスをコードクローンメトリクスで代用するのは難しい.

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て，“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．

また，RQ3 に対する答えから，周辺コードへの依存性である外部要素の個数や，そのうち，あるクローンセット内のコードクローン内の変数からの参照が一致していないということが，従来のクローンメトリクスである LEN，NIF，POP，RAD，RNR からは説明できなかった．そのため，コードクローンや周辺コードを表示するだけではなく，外部要素の個数やそのうち，差分となっている個数を容易に確認できる必要があると考えられる．

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　周辺コード調査用ツールの試作

3 章での結果から，コードクローンだけではなく，周辺コードを調べることの有用性がわかった．しかしながら，周辺コードを調査するためのツールや，周辺コードの差分についての情報を表示するツールは存在していない．

そこで，Java で記述されたソースコードを解析し，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試作した．本ツールは，Eclipse のプラグインとして実装されている．試作したコードクローン周辺コード調査用ツールのスクリーンショットを図 25 に示す．

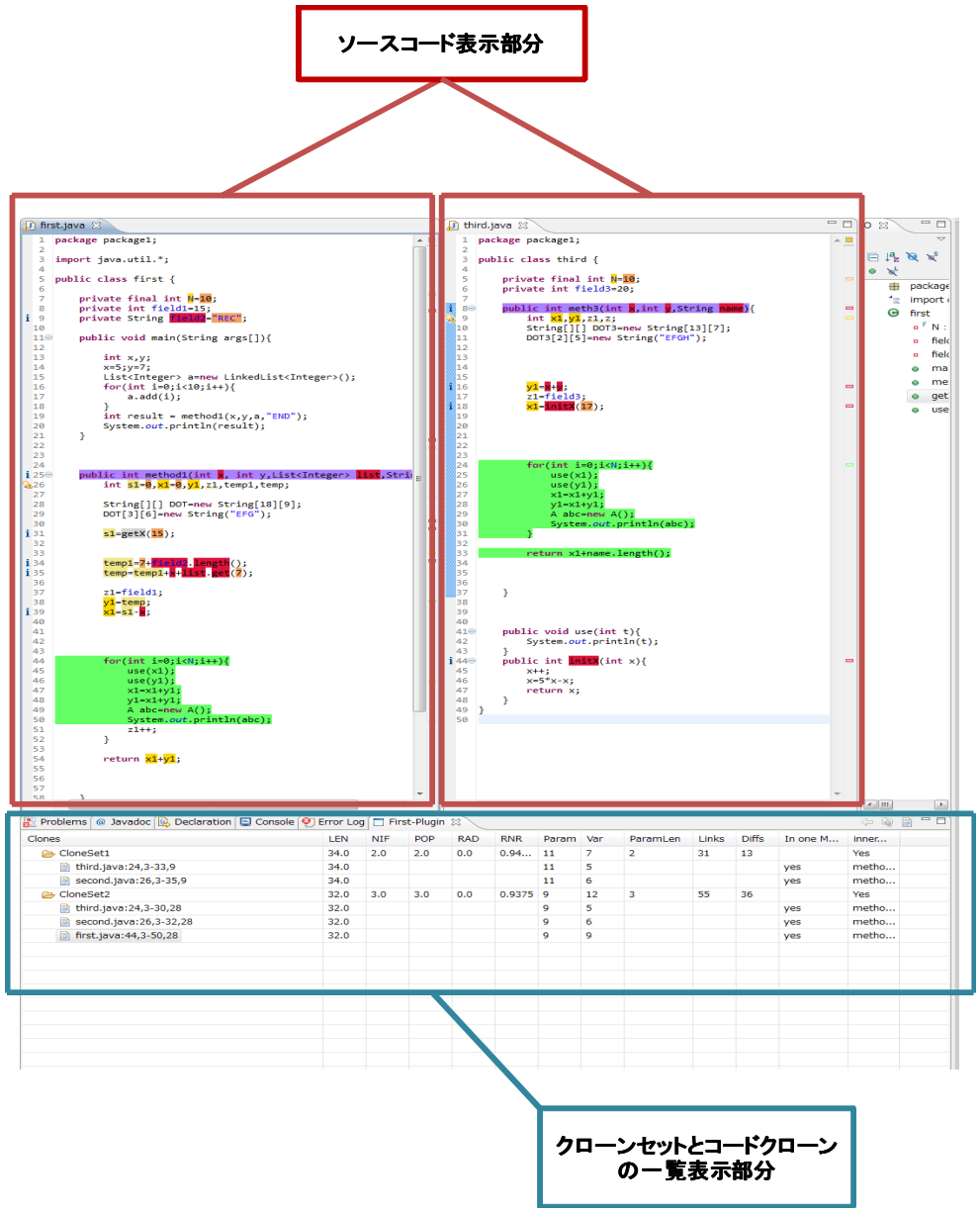


図 25: コードクローンの周辺コードを解析し，可視化する Eclipse のプラグイン

プラグインはソースコード表示部分とクローンセット，コードクローンの一覧表示部分から構成されている．以下にそれぞれの部分について説明する．

図 26 にクローンセットとコードクローンの一覧表示部分（クローンセット，コードクローン，それに対応するメトリクスの値の一覧表示）のスクリーンショットを示す．

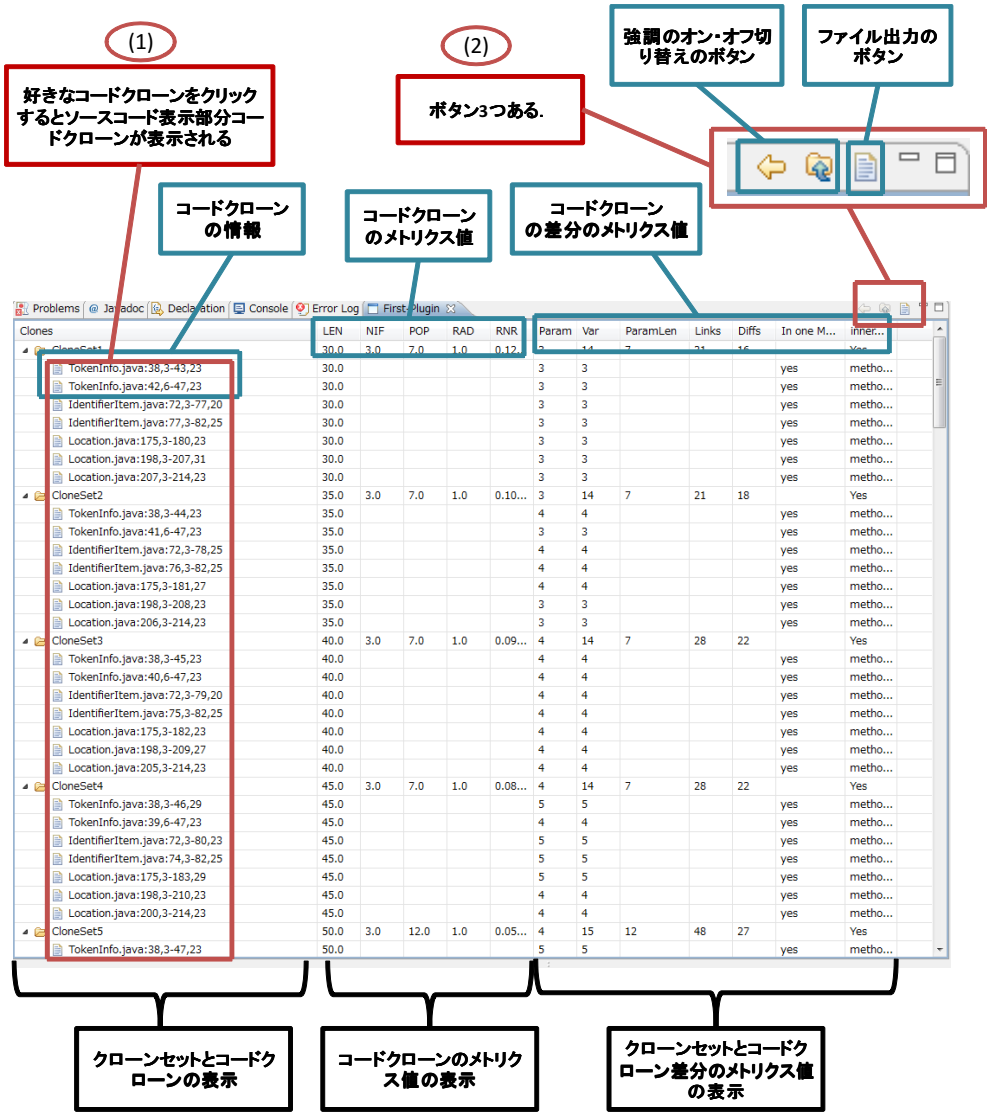


図 26: 一覧表示する処理の実装

図 26 の “コードクローンの情報 ” を取得するために，本研究では，字句解析ベースのコードクローン検出ツール CCFinder を利用した．

図 26“コードクローンの差分のメトリクス ” を取得するために，まずコードクローンの周辺コードを特定した．コードクローンの周辺コードを取得するために，以下の処理を行った．

1. Eclipse の ASTParser を使って，コードクローンを含むファイルの抽象構文木（AST）を取得する．

2. ビジターパターンによってツリーウォークする Eclipse の ASTVisitor クラスを利用し

て，取得した AST からコードクローン内に出現する変数を取得する．

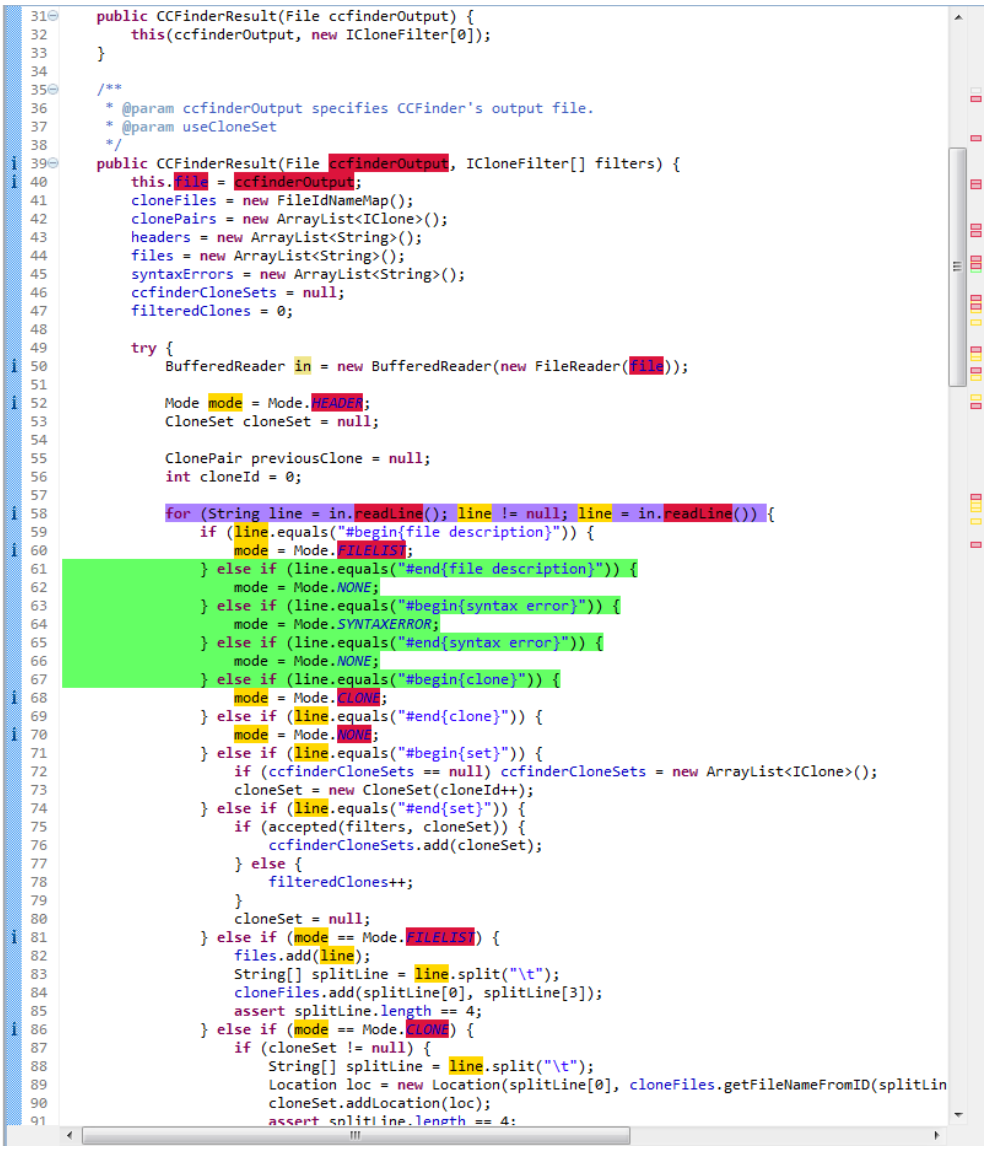
3. 取得したコードクローンの各変数に対するデータフロー上に出現する変数とメソッドを Eclipse の ASTVisitor クラスを繰り返し利用して取得する．

4. 最内制御ブロックを Eclipse の AST を利用して，コードクローン内の変数あるいはリテラルノードから親ノードをたどって取得する．

取得した周辺コードを解析して，外部要素の差分を特定し，特定した外部要素とその差分から“コードクローンの差分のメトリクス”を計算する．

図 26 のクローンセットとコードクローンの表示から選びたいコードクローンのところをクリックすれば，そのコードクローンが含まれるファイルがソースコード表示部分（図 27）に開かれる (図 26 の (1))．図 26 の右上には 3 つのボタンがある (図 26 の (2))．左のボタンはソースコード表示部分に表示されたファイル上のコードクローン周辺コードのデータフローの強調のオンオフを切り替えるボタンである．中央のボタンはソースコード表示部分に表示されたファイル上のコードクローンの外部要素の差分の強調のオンオフを切り替えるボタンである．右側のボタンは図 26 に表示しているメトリクス値を各クローンセットの番号と対応するメトリクス値が記載されたファイル，各コードクローンの番号と対応するメトリクス値が記載されたファイル，メソッドに含まれていないコードクローンの個数と変数を持たないコードクローンの個数が記載されたファイルを出力する．

図 27 に図 26 の一覧表示から選択されたコードクローンをソースコード表示部分に表示する実装を示す．



```
31    public CCFinderResult(File ccfinderOutput) {
32        this(ccfinderOutput, new ICloneFilter[0]);
33    }
34
35    /**
36     * @param ccfinderOutput specifies CCFinder's output file.
37     * @param useCloneSet
38     */
39    public CCFinderResult(File ccfinderOutput, ICloneFilter[] filters) {
40        this.file = ccfinderOutput;
41        cloneFiles = new FileIdNameMap();
42        clonePairs = new ArrayList<IClone>();
43        headers = new ArrayList<String>();
44        files = new ArrayList<String>();
45        syntaxErrors = new ArrayList<String>();
46        ccfinderCloneSets = null;
47        filteredClones = 0;
48
49        try {
50            BufferedReader in = new BufferedReader(new FileReader(file));
51
52            Mode mode = Mode.HEADER;
53            CloneSet cloneSet = null;
54
55            ClonePair previousClone = null;
56            int cloneId = 0;
57
58            for (String line = in.readLine(); line != null; line = in.readLine()) {
59                if (line.equals("#begin{file description}")) {
60                    mode = Mode.FILELIST;
61                } else if (line.equals("#end{file description}")) {
62                    mode = Mode.NONE;
63                } else if (line.equals("#begin{syntax error}")) {
64                    mode = Mode.SYNTAXERROR;
65                } else if (line.equals("#end{syntax error}")) {
66                    mode = Mode.NONE;
67                } else if (line.equals("#begin{clone}")) {
68                    mode = Mode.CLONE;
69                } else if (line.equals("#end{clone}")) {
70                    mode = Mode.NONE;
71                } else if (line.equals("#begin{set}")) {
72                    if (ccfinderCloneSets == null) ccfinderCloneSets = new ArrayList<IClone>();
73                    cloneSet = new CloneSet(cloneId++);
74                } else if (line.equals("#end{set}")) {
75                    if (accepted(filters, cloneSet)) {
76                        ccfinderCloneSets.add(cloneSet);
77                    } else {
78                        filteredClones++;
79                    }
80                    cloneSet = null;
81                } else if (mode == Mode.FILELIST) {
82                    files.add(line);
83                    String[] splitLine = line.split("\t");
84                    cloneFiles.add(splitLine[0], splitLine[3]);
85                    assert splitLine.length == 4;
86                } else if (mode == Mode.CLONE) {
87                    if (cloneSet != null) {
88                        String[] splitLine = line.split("\t");
89                        Location loc = new Location(splitLine[0], cloneFiles.getFileNameFromID(splitLin
90                        cloneSet.addLocation(loc);
91                        assert splitLine.length == 4;
```

図 27: ファイル上にコードクローン部分，その周辺コード部分，差分を強調して表示する処理の実装

まず，クローンセットとコードクローンの一覧表示部分から選択されたコードクローンが存在するファイルを Eclipse の JDT エディタ上に開く．JDT エディタ上で開く際，コードクローン部分とコードクローンの周辺コード（データフロー全体と最内制御ブロック）と外部要素の差分を異なる色でハイライトしてソースコードを表示する．コードクローン部分を緑色，コードクローンに存在する変数を橙色，コードクローンに存在する変数のデータフ

ローの変数を薄茶色，リテラルを薄赤色でハイライトする．コードクローンの周辺コードに対する差分を濃赤色でハイライトする．

5　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 Lingxiao ら [3] の研究がある．Lingxiao 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，周辺コードの不一致の分類などを定義している．Lingxiao 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．

Lingxiao 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）コードクローンの周辺コードを解析を行い，周辺コード対する調査をしている．そして，その調査の結果に基づいて，開発者がコードクローン間の周辺コードの理解と比較に利用できるコードクローンとその周辺コードを可視化するツールを試作した．

ZhenChang ら [21] 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 PDG 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．本研究はコードクローンの周辺コードを比較するのに対し，ZhenChang らはコードクローンの中身も比較している．

6　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，また，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，それらは従来のコードクローンメトリクスでは説明することが難しいということがわかった．また，調査に基づき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試作した．本ツールでは周辺コードについての情報を数値と視覚の両面で表現している．

今後の課題として，試作したツールを実際のソフトウェア開発に適用して評価することが挙げられる．また，本調査では主にタイプ 2 クローンに対応する CCFinder を用いてコードクローンを検出したが，タイプ 3 クローンを対象としたその他のコードクローン検出ツールを用いて，周辺コードの評価を行うことも重要な課題である．

謝辞

本研究において，常に適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします.

本研究において，随時適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に心より深く感謝いたします.

本研究において，逐次適切な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に心より深く感謝いたします.

本研究において，終始適切な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻眞鍋雄貴特任助教に心より深く感謝いたします.

本研究において，様々な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻伊達浩典氏に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深く感謝いたします.

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] Cory J. Kapser and Michael W. Godfrey. “cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering*, Vol. 13, No. 6, pp. 645–692, 2008.

[3] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 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] Jens Krinke. Is cloned code older than non-cloned code? *IWSC '11 Proceedings of the 5th International Workshop on Software Clones*.

[5] Jan Harder and Nils Gode. Cloned code: stable code. *Journal of Software: Evolution and Process*.

[6] S. Bellon, R. Koschke, G. Antoniola, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transaction Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[7] I.D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. *Proc. International Conference on Software Maintenance '98*, pp. 368–377, 1998.

[8] C.K. Roy and J.R. Cordy. A survey on software clone detection research. Technical report, Queen's University at Kingston, 2007.

[9] R.Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007.

[10] S.Jarzabek and S.Li. Eliminating redundancies with a "composition with adaptation" meta-programming technique. *ESEC/FSE*, pp. 96–105, 2003.

[11] D.C.Rajapakse and S.Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. *ICSE*, pp. 116–126, 2007.

[12] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016–1038, 2002.

[13] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 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] H.A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 156–165, 2005.

[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. *Proc. International Conference on Software Maintenance '96*, pp. 244–253, 1996.

[16] K. Kontogiannisa, R. DeMori, E. Merlo, M. Galler, 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] T. Kamiya, S. Kusumoto, and K. Inoue. 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] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. 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] Takashi Ishio, Shogo Etsuda, and Katsuro Inoue. A lightweight visualization of interprocedural data-flow paths for source code reading, 2012.

[20] 工藤良介. コードクローンに含まれるメソッド呼び出しの変更度合の分析. Master's thesis, 大阪大学修士論文, 2013.

[21] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. Clonedifferentiator: Analyzing clones by differentiation. *ASE '11 Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 576–579, 2011.