社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

コードクローン間の類似度に基づく無害なコードクローンの自動判定手法

土居　真之† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
E-mail: †{m-doi,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　互いに類似するコード片であるコードクローンの存在はソフトウェア保守に悪影響を与えるとされている．しかし全てのコードクローンが必ずしもソフトウェアの保守に有害であるとは限らない．そのためコードクローン検出器によって検出されたコードクローンに対して，コードクローンが有害か否かを判定する必要がある．この有害なコードクローンを判定する手法として，機械学習を用いる手法が既存研究で提案されている．これは開発者による有害か無害かの判定結果を学習することで，クローンが有害か否かを判定する手法である．しかし機械学習を用いるためには学習用のデータセットを用意しなくてはならない．既存手法では学習用データセットとして開発者やプロジェクトごとに有害か否かの判断結果を収集する必要があるため，準備にかかるコストが高いという問題がある．一方でクローンセットの中には開発者やプロジェクトによらず無害であることが自明なクローンセットが存在する．無害であることが自明なクローンセットは学習用データセットを構築する際に自動で無害と判断することで準備にかかるコストを削減できる．そこで本研究では，クローンセット間の類似度を可視化することでコードクローンを分類する手法を提案する．さらに提案手法はツール HarmfulCloneClassifier として実装した．実験の結果 3,993 種類のクローンセットから 145 個の 言語固有のクローンと 507 個の プロジェクト固有のクローンを抽出することができ，抽出された言語固有のクローンは無害であることを確認した．

キーワード　コードクローン，ソフトウェア保守，機械学習

1. ま え が き

コードクローン (以降，クローンと呼ぶ) とは，ソースコード中の同一あるいは類似するコード片のことを指す．このクローンはソフトウェアの保守に悪影響を与えるとされており，今まで多くの研究が行われている．中でもクローンの検出技術，及びその除去技術はさかんに研究されており，クローンをソースコード中から自動的に検出するツールも多数開発，公開されている [1]．このクローン検出ツールは対象となるソフトウェア中から特定したクローンをすべてを出力するため，出力されるクローンの量は膨大になる．しかしこれらのクローンすべてがリファクタリングすべき有害なクローンとは限らない．例えばクローンの一部分がよく編集されるため将来的にクローンではなくなる場合はリファクタリングの必要がないとされている [2], [3]．このため，開発者は検出ツールによって提示された膨大なクローンの中からリファクタリングが必要な有害なクローンを特定する必要があり，これには非常に多くの作業を要する．さらに，あるクローンが有害か否かを判別する基準は，個々の開発者によって異なる．したがって，どのようなクローンがソフトウェア保守に有害かを一般化することは難しく，有害か否かの判定は困難である．

この問題に対し，Yang らは機械学習を用いることでクローンセットが有害か否かを判定する手法を提案している [4]．この手法は複数人の開発者に検出されたクローンセットの一部を有害か判定してもらい，その結果を学習することで残りのクローンセットが有害であるか判定するというものである．しかしクローンセットが有害であるかの基準はプロジェクトやその開発者によって異なるため，プロジェクトの開発者に判定してもらう必要がある [5]．そのため，学習用のデータセットを構築するのに膨大なコストがかかる．一方でどのプロジェクトにおいても無害であることが自明なクローンセットが存在する．例えば言語の仕様から生じる定型処理のクローンは長期間クローンとして存在しやすいため，ソフトウェア開発に悪影響を及ぼさない無害なクローンである [3]．無害であることが自明なクローンセットを事前に取り除くことができれば，開発者による判断が必要なクローンセットを削減することができる．また言語の仕様から生じる定型処理のクローンはプロジェクトによらず似た処理であるため，複数プロジェクトにまたがって類似した記述がされやすいと考えられる．そのため複数プロジェクトから別々にクローンを検出し，検出されたクローンセット間の類似度を用いることで，無害であることが自明なクローンセットと判断が必要なクローンセットを分類することが可能であると著者らは考えた．

そこで本研究ではクローンセットを自動で分類する手法を提

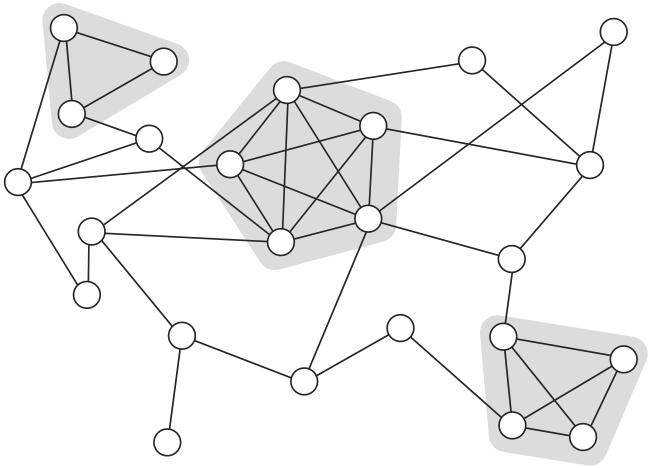


図 1　クリークの例

案する．自動分類はクローンセット間の類似度に着目したグラフを作成し，クリークを形成するか否かで分類する．一つのクローンセットをグラフの一頂点とし，クローンセット間の類似度が閾値以上の場合辺を存在させる．これにより類似しているクローンセットは辺で結ばれ，互いに類似し合っているクローンセット間でクリークを形成する．複数プロジェクトにまたがってクリークを形成するクローンは類似した定型処理のクローンであると考えられるため，無害なクローンとみなすことが可能となる．提案手法の評価のために実際の 4 プロジェクトのクローンセットに対して提案手法を適用した．その結果 3,993 個のクローンセットから 145 個の 言語固有のクローンと 507 個のプロジェクト固有のクローンを抽出することができた．また抽出された 言語固有のクローンの一部を確認するとコンストラクタや多くのクラスで定義される処理など無害な処理が分類されていることを確認した．さらにこのクローンセット間の類似度を基にしたグラフを描画するツール HarmfulCloneClassifier を作成した．

2. 準　　備

この章では，本論文で使用する語句，技術を説明する．

2.1 クリーク

部分グラフのうち完全グラフになっているものをクリークと呼ぶ．クリークは密な構造を表す最も基礎的な構造である．3 頂点以上のクリークの例を図 1 に示す．図 1 において，背景色が異なる 3 つの部分グラフがクリークである．

2.2 クローンセット間の類似度

クローンセット間の類似度は Yang らの手法 [4] で用いられている TF-IDF 値 (Term Frequency - Inverse Document Frequency) とする [6] ．以降の説明では，1 つのクローンセットを d，クローンセットの集合を D と表記する．

TF-IDF 値の計測に際し，まずクローンセット d を構成するコード片のトークン型列から N-gram を抽出する．例えば，図 2(a) のソースコードから得られるトークン型列 (図 2(b)) に対して N = 3 とすると，図 2(c) のような N-gram が得られる．コード片のトークン型列を用いることで，変数名やリテラル等の違いを吸収することが可能となり，算出するクローンセット間の類似度，及び最終的な予測精度の向上が期待できる．N-gram の N のサイズは大きいほど精度が高くなるが，計算時間が膨大となる．

```
public void updateCrosshair(double transX, int dataIndex) {
        if (this.anchor == null) {
                return;
        }
        double d = Math.abs(transX);
        if (d < this.distance) {
                this.dataIndex = dataIndex;
                this.distance = d;
        }
}
```

(a) ソースコード

```
PUBLIC VOID IDENTIFIER LPAREN DOUBLE IDENTIFIER COMMA INT
IDENTIFIER RPAREN LBRACE
        IF LPAREN THIS DOT IDENTIFIER EQUALEQUAL NULL RPAREN
LBRACE
                RETURN SEMICOLON
        RBRACE
        DOUBLE IDENTIFIER EQUAL IDENTIFIER DOT IDENTIFIER
LPAREN IDENTIFIER RPAREN SEMICOLON
        IF LPAREN IDENTIFIER GREATER THIS DOT IDENTIFIER RPAREN
LBRACE
                THIS DOT IDENTIFIER = IDENTIFIER SEMICOLON
                THIS DOT IDENTIFIER = IDENTIFIER SEMICOLON
        RBRACE
RBRACE
```

(b) トークン型列

```
PUBLIC VOID IDENTIFIER
        VOID IDENTIFIER LPAREN
                IDENTIFIER LPAREN DOUBLE
                        LPAREN DOUBLE IDENTIFIER
                                DOUBLE IDENTIFIER COMMA …
```

(c) N-gram

図 2　字 句 解 析

得られた N-gram のうちの 1 つを t と表記する．次に，得られたすべての N-gram に対して term - frequency $tf(t,d)$ を式 (1) を用いて計測する．

$$tf(t,d) = \frac{|t : t \in d|}{|d|} \tag{1}$$

続いて，inverse document frequency $idf(t,D)$ を式 (2) を用いて計測する．$idf(t,D)$ は t がどれだけ多くのクローンセット内に出現するのかを表す指標であり，多くのクローンセット内に出現している場合にその値が小さくなる．

$$idf(t,D) = \log \frac{|D|}{|d \in D : t \in d|} \tag{2}$$

次に，式 (1) 及び式 (2) を用いて，$tfidf$ を計測する (式 (3)，式 (4))．

$$tfidf(t,d,D) = tf(t,d) \times idf(t,D) \tag{3}$$

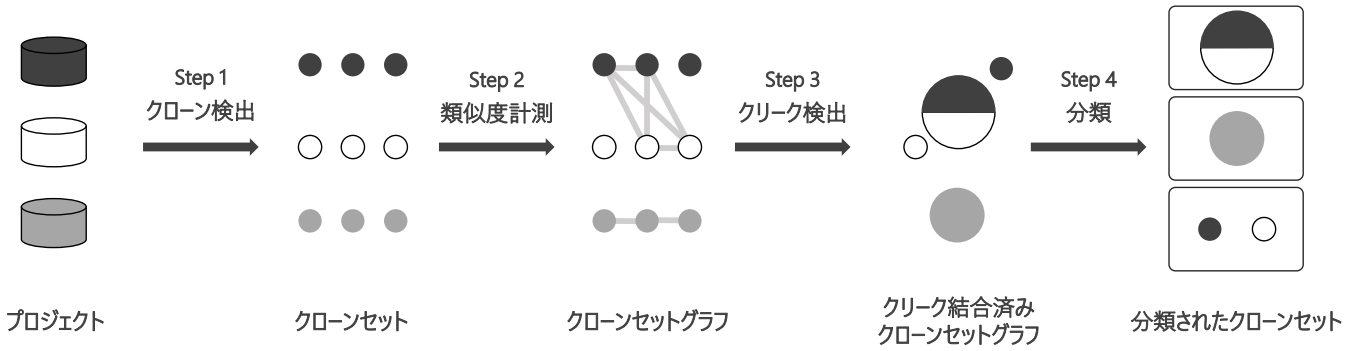


図 3　提案手法の概要

```
public CopyOneFile(CopyOneFile other, DataInput in) {
        this.in = in;
        this.dest = other.dest;
        this.name = other.name;
        this.out = other.out;
        this.tmpName = other.tmpName;
        this.metaData = other.metaData;
        this.bytesCopied = other.bytesCopied;
        this.bytesToCopy = other.bytesToCopy;
        this.copyStartNS = other.copyStartNS;
        this.buffer = other.buffer;
}
```

図 4　言語固有のクローンの例

$$\overrightarrow{tfidf(d,D)} = [tfidf(t,d,D) \quad \forall t \in d] \tag{4}$$

$tfidf(t,d,D)$ は個々の N-gram に対する TF-IDF 値を表しており，$\overrightarrow{tfidf(d,D)}$ は $tfidf(t,d,D)$ を連結して作成したベクトルである．この $\overrightarrow{tfidf(d,D)}$ を用いて，2 つのクローンセット a，b 間の類似度 $sim(a,b,D)$ を以下のように計測する．

$$sim(a,b,D) = \overrightarrow{tfidf(a,D)} \cdot \overrightarrow{tfidf(b,D)} \tag{5}$$

2.3　クローンセットの分類

本研究ではクローンセットを以下の 3 種類に分類する．

- 言語固有のクローン
- プロジェクト固有のクローン
- 孤立したクローン

これらの分類について説明する．

言語固有のクローンはソースコードは言語の仕様や構文からクローンとなりやすいソースコードである．そのため，言語固有のクローンと分類されたクローンはクローンとなっても問題のない無害なクローンであると筆者らは考える．言語固有のクローンの例を図 4 に示す．図 4 のようなコンストラクタでメンバ変数に代入する処理は多くのプロジェクトで用いられていると考えられ，リファクタリングの価値も低く無害なクローンである．

一方プロジェクト固有のクローンに分類されるクローンのソースコードはプロジェクト固有の処理やコード規約からクローンとなりやすいソースコードである．そのためプロジェクト固有のクローンと分類されたクローンはプロジェクトの開発者による判断が必要なクローンであると筆者らは考える．プロジェクト固有のクローンの例を図 5 に示す．図 5 はバイトコードを編集してメンバ変数の配列に代入する処理であり，バイトコードを編集するプロジェクト固有の処理である．そのためプロジェクトや開発者によって判断が異なると考えられえる．

```
if (counter != 0) {
        this.contents[annotationsOffset++] = (byte) (counter >> 8);
        this.contents[annotationsOffset++] = (byte) counter;
        int attribute = this.contentsOffset - attributeOffset - 4;
        this.contents[attributeOffset++] = (byte) (attribute >> 24);
        this.contents[attributeOffset++] = (byte) (attribute >> 16);
        this.contents[attributeOffset++] = (byte) (attribute >> 8);
        this.contents[attributeOffset++] = (byte) attribute;
        attributesNumber++;
}
```

図 5　プロジェクト固有のクローンの例

最後にこれら 2 つの分類に分類されないクローンセットを孤立したクローンに分類する．孤立したクローンに分類されるクローンのソースコードは他に類似したクローンセットが存在しないソースコードである．そのため，有害なクローンセットである可能性が高いと筆者らは考える．

3. 提 案 手 法

本研究ではクローンセット間の類似度に基づいたグラフからクローンセットを分類する手法を提案する．提案手法の概要を図 3 に示す．提案手法における入力はプロジェクトであり，出力は分類された各プロジェクトのクローンセットである．

提案手法は 4 つの Step で構成されている．

Step 1　クローンの検出
Step 2　クローンセット間の類似度計測
Step 3　クリークの検出
Step 4　クローンセットの分類

以降，各 Step について説明する．

Step 1 では入力されたプロジェクトごとにクローンセットを検出する．Step 2 では Step 1 で検出されたクローンセットに対し 2.2 で述べた計測方法でクローンセット間の類似度を計測し，類似度を基にグラフを作成する．作成されるグラフの頂点は各クローンセットを表し，グラフの辺はクローンセット間の類似度が一定値以上の場合存在する．Step 3 では Step 2 で作成したグラフからクリークを検出し，検出されたクリークを一つの頂点に結合してグラフを再描画する．再描画されるグラフ

では頂点の面積比が頂点内に存在するクローンセットの数に対応している．最後に Step 4 でグラフにおける各頂点のクローンセットを 2.3 で述べた分類に分類する．分類はクリークである頂点のうち，複数プロジェクトによって構成されているクリークは言語固有のクローン，単一プロジェクトのみで構成されているクリークはプロジェクト固有のクローンに分類する．残りのクリークを形成していない頂点は孤立したクローンに分類する．

4. 実　　装

提案手法をツール HarmfulCloneClassifier として実装した．本章では提案手法の各 Step の実装と HarmfulCloneClassifier について説明する．

4.1 クローンの検出

クローンの検出には既存のクローン検出器を用いる．クローンの検出器は多数存在するが，本研究では NiCad [7] を用いた．NiCad により検出するクローンの単位はブロック単位とし，クローン間の類似度の閾値はデフォルトの 0.30 として検出を行った．

4.2 クローンセット間の類似度計測

2.2 で述べたクローンセット間の類似度を計測する．トークン型列の生成には Eclipse JDT を用いた．またトークン型列から作成する　N-gram　の N のサイズは $N = 5$ として構築した．さらに計測された類似度に基づいてグラフを作成する．グラフの頂点は各クローンセットであり，辺はクローンセット間の類似度が閾値以上の場合存在する．この類似度の閾値は 0.50 とした．

4.3 クリークの検出

クリークの検出にはクリーク列挙器である PCE を用いる [8]．PCE では検出するクリークの最小頂点数を指定することができる．本研究ではこの最小頂点数を 3 としてクリークを検出する．

4.4 クローンセットの分類

クリークの結合されたグラフからクローンセットを 2.3 で述べた分類に分類する．複数プロジェクトによって構成されているクリークは言語固有のクローン，単一プロジェクトのみで構成されているクリークはプロジェクト固有のクローンに分類する．クリークを形成していない残りの頂点は孤立したクローンに分類する．

表 1　実験対象プロジェクト

プロジェクト名	# クローンセット
JFreeChart	354
Eclipse.jdt.core	2,003
Lucene	1,012
Tomcat	624

4.5 HarmfulCloneClassifier

クローンセット間の類似度可視化ツールを実装し， HarmfulCloneClassifier と名付けた．HarmfulCloneClassifier はクローンセット間の類似関係を force-directed アルゴリズム [9] を用いてグラフに描画するブラウザアプリケーションである．force-directed アルゴリズムはそれぞれの辺をバネとみなし，頂点をクーロンの法則に従わせるアルゴリズムである．またグラフの頂点にカーソルを合わせることで頂点内のクローンセットのソースコードを見ることができる．これによりグラフを構成するクローンセットのコード片を確認することができる．HarmfulCloneClassifier のグラフ描画には JavaScript ライブラリである D3 を用いた．

5. 実　　験

本章では提案手法の評価のために行った実験について説明する．

5.1 実験対象

実験対象のプロジェクトは異なるドメインから JFreeChart，Eclipse.jdt.core，Lucene，Tomcat の 4 つの Java プロジェクトを用いた．用いた各プロジェクトのリビジョンは 2019/1/28 時点で最新のリビジョンである．前準備として各プロジェクトからテストコードを排除したのち NiCad でクローンを検出した．検出されたクローンセットの数を表 1 に示す．この合計 3,993 個のクローンセットを提案手法を用いて分類した．

5.2 実験結果

分類の結果を表 2 に示す．全てのプロジェクトにおいて孤立したクローンが最も多く分類されており，各プロジェクトにおいて 73.7%-89.1% のクローンセットが孤立したクローンに分類された．また 4 つのプロジェクトのうち Tomcat を除く 3 プロジェクトについては言語固有のクローンよりもプロジェクト固有のクローンが多く分類された．

次に HarmfulCloneClassifier によって描画された全体のグラフを図 6 に示す．グラフの頂点の色は頂点内のクローンセットが存在するプロジェクトを表している．また頂点の面積比は頂点内に存在するクローンセットの数に対応している．頂点内の色の面積比がそれぞれのクローンセットが存在するプロジェクトの割合を示している．グラフから孤立したクローンが多く点在しており，言語固有のクローンとプロジェクト固有のクローンの頂点ではグラフを形成していることがわかる．図 7 は言語固有のクローンとプロジェクト固有のクローンの頂点を含むグラフを拡大したものである．言語固有のクローンの頂点を含むグラフである図 7(a) では，言語固有のクローン以外にも様々

表 2　クローンの分類結果

クローンの分類	JFreeChart	Eclipse.jdt.core	Lucene	Tomcat	合計
#言語固有のクローンセット	16 (4.5%)	48 (2.4%)	38 (3.7%)	43 (6.9%)	145 (3.6%)
#プロジェクト固有のクローンセット	77 (21.8%)	298 (14.9%)	107 (10.6%)	25 (4.0%)	507 (12.7%)
#孤立したクローンセット	261 (73.7%)	1,657 (82.7%)	867 (85.7%)	556 (89.1%)	3,341 (83.7%)

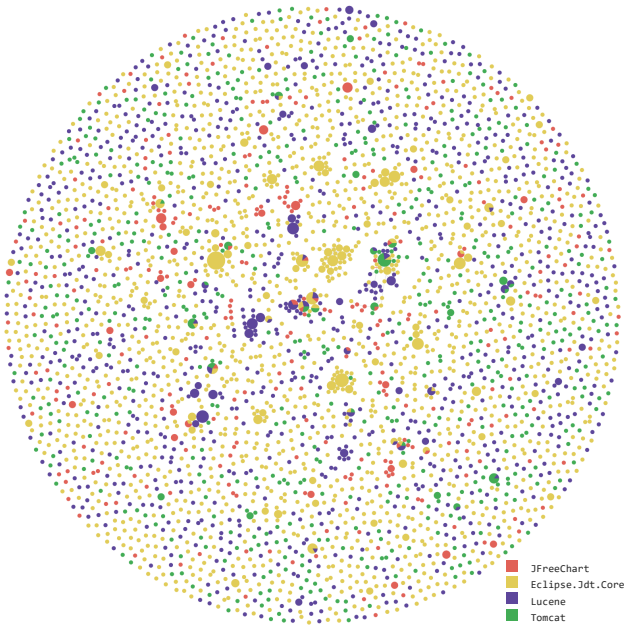
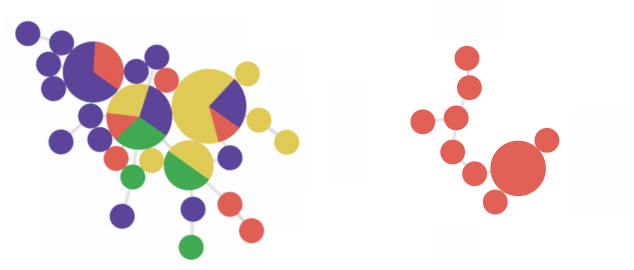


図 6　クローンセットグラフ



(a) 言語固有のクローンと分類されたクローンセットを含むグラフ　(b) プロジェクト固有のクローンと分類されたクローンセットを含むグラフ

図 7　言語固有のクローンと分類されたクローンの例

なプロジェクトの孤立したクローンとグラフを形成している．一方プロジェクト固有のクローンの頂点を含むグラフである図 7(b) のグラフは，プロジェクト固有のクローンと同じプロジェクトでグラフを形成しているものが多い．

6. 考　　察

6.1　言語固有のクローンと分類されたクローン

分類結果のうち JFreeChart から言語固有のクローンと分類された 16 のクローンセットを全て目視確認をした．16 セットのうち最も多かったのはコンストラクタのクローンセットと多くのクラスで定義されるメソッドのクローンセットであり，それぞれ 4 セット存在した．それぞれのクローンセットのソースコード片の例を図 8 に示す．

JFreeChart のクローンセットのうち言語固有のクローンと分類されたコンストラクタの例を図 8(a) に示す．このコード片は図 7(a) のグラフ内に存在するコード片である．コンストラクタはクラスのインスタンス生成時に実行されるメソッドであり，そのクラスのメンバ変数を初期化する処理を行うことが多い．メンバ変数を初期化する処理はプロジェクトに関わらず

jfreechart/src/main/java/org/jfree/chart/renderer/xy/CandlestickRenderer.java

```
public CandlestickRenderer(double candleWidth, boolean
drawVolume, XYToolTipGenerator toolTipGenerator) {
        super();
        setDefaultToolTipGenerator(toolTipGenerator);
        this.candleWidth = candleWidth;
        this.drawVolume = drawVolume;
        this.volumePaint = Color.gray;
        this.upPaint = Color.green;
        this.downPaint = Color.RED;
        this.useOutlinePaint = false;
}
```

(a) コンストラクタのクローン

jfreechart/src/main/java/org/jfree/chart/annotations/AbstractXYAnnotation.java

```
public boolean equals(Object obj) {
        if (obj == this) {
                return true;
        }
        if (!(obj instanceof AbstractXYAnnotation)) {
                return false;
        }
        AbstractXYAnnotation that = (AbstractXYAnnotation) obj;
        if(!ObjectUtils.equal(this.toolText, that.toolText)) {
                return false;
        }
        if (!ObjectUtils.equal(this.url, that.url)) {
                return false;
        }
        return true;
}
```

(b) 多くのクラスで定義されるメソッドのクローン

図 8　言語固有のクローンと分類されたクローンの例

代入文が連続することが多いため言語固有のクローンと分類された．またコンストラクタの処理はメンバ変数の追加や削除により頻繁に編集されるが，代入文が追加削除されるだけであることが多いためリファクタリングの価値が低い無害なクローンであると考えられる．

次に言語固有のクローンと分類された多くのクラスで定義されるメソッドの例を図 8(b) に示す．これは equals(Object obj) というメソッドで，引数で与えられたオブジェクトがメソッドが定義されているオブジェクトと等しいかを判定するメソッドである．equals(Object obj) は java.lang.Object で宣言されているメソッドである．java.lang.Object は全てのクラスのスーパークラスであり，複数プロジェクトで再定義するクラスが存在するため言語固有のクローンと分類された．他に言語固有のクローンと分類された java.lang.Object のメソッドには toString() や hashCode() が存在した．またこのような多くのクラスで定義されるメソッドは構造が単純であり，編集されることも少ないため無害なクローンであると考えられる．

6.2　プロジェクト固有のクローンと分類されたクローン

分類結果のうち JFreeChart からプロジェクト固有のクローンと分類されたクローンセットを目視確認をした．分類されたクローンの例を図 9 に示す．このコード片は図 7(b) のグラフ内に存在するコード片である．このコード片ではグラフの目

jfreechart/src/main/java/org
/jfree/chart/renderer/xy/CandlestickRenderer.java

```
if (isTickLabelsVisible()) {
        g2.setFont(getTickLabelFont());
        AxisState state = new AxisState();
        // we call refresh ticks just to get the maximum width or height
        refreshTicks(g2, state, plotArea, edge);
        if (edge == RectangleEdge.TOP) {
                tickLabelHeight = state.getMax();
        }
        else if (edge == RectangleEdge.BOTTOM) {
                tickLabelHeight = state.getMax();
        }
        else if (edge == RectangleEdge.LEFT) {
                tickLabelWidth = state.getMax();
        }
        else if (edge == RectangleEdge.RIGHT) {
                tickLabelWidth = state.getMax();
        }
}
```

図 9　分類されたクローンの例

盛りの最大値を取得する処理を行っており，変数 edge の値によって複数の条件分岐を行っている．この条件分岐の記述は他の JFreeChart のクローンセットにも存在していたが，他のプロジェクトからは見受けられなかった．したがってこの条件分岐の記述はプロジェクト固有の記述であると考えられる．このようにプロジェクト固有のクローンと分類されたクローンはプロジェクト固有の処理が記述されている傾向があると考えられる．一方で今回の実験で用いた 4 プロジェクトでグラフ描画を扱っているプロジェクトはこの JFreeChart のみである．言語固有のクローンに多くのクラスで定義されるメソッドが分類される傾向があることから，JFreeChart 以外のグラフ描画を扱うプロジェクトを実験対象に追加した場合，多くのクラスで定義される処理として言語固有のクローンと分類される可能性もあると考えられる．

7. 妥当性への脅威

コードクローンの検出には NiCad を用いた．しかし他の検出器を用いた場合，異なるクローンセットが検出され本実験とは異なる結果になる可能性がある．また NiCad の類似度の閾値は 0.30 として実験をした．そのためクローン間で構文に差異が存在する場合があり，クローンセットの N-gram が不適切に抽出され類似度の計算が正しく行えていない可能性がある．しかし不適切な抽出される N-Gram は十分に少なく，類似度の計算に与える影響は十分に低いと考える．

提案手法を評価するために 4 つの Java プロジェクトで実験した．しかし他の Java プロジェクトや Java 以外の言語のプロジェクトに適応した場合，本実験とは異なる結果になる可能性がある．

分類結果を確認するために JFreeChart についてのみ確認を行った．その結果言語固有のクローンにはコンストラクタや多くのクラスで定義される処理が多く，プロジェクト固有のクローンにはプロジェクト固有の処理が存在した．しかし他の 3 プロジェクトに関しても同様のことは言えない可能性がある．

8. あ と が き

本研究ではクローンセット間の類似度に基づいたグラフからクローンセットを分類する手法を提案した．提案手法を 4 つのプロジェクトに対して適応した結果， 3,993 種類のクローンセットから 145 個の 言語固有のクローンと 507 個の プロジェクト固有のクローンを抽出することができた．また抽出された言語固有のクローンの一部を確認するとコンストラクタや多くのクラスで定義されるメソッドなど言語固有な実装が分類されていることを確認した．

今後の課題としては次のようなものが考えられる．

有害なクローンの評価実験

提案手法によって各分類に分類されたクローンセットに対し，有害なクローンと分類の関係を評価を行うことを検討している．

クローンセット情報の可視化

本研究では頂点の色はプロジェクトで統一した．しかし他にもクローンセット情報を表示することで新たな傾向がわかる可能性がある．例えばクローンセットのトークン数で色分けをすることが挙げられる．

謝辞　本研究は日本学術振興会科学研究費補助金基盤研究 (B)(課題番号：17H01725) の助成を得て行われた．

文　　献

[1] A. Sheneamer and J. Kalita, “A survey of software clone detection techniques,” International Journal of Computer Applications, vol.137, no.10, pp.1–21, 2016.

[2] W. Wang and M.W. Godfrey, “Recommending clones for refactoring using design, context, and history,” 2014 IEEE International Conference on Software Maintenance and Evolution, pp.331–340, Sep. 2014.

[3] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” SIGSOFT Softw. Eng. Notes, vol.30, no.5, pp.187–196, Sept. 2005.

[4] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Classification model for code clones based on machine learning,” Empirical Softw. Engg., vol.20, no.4, pp.1095–1125, Aug. 2015.

[5] A. Charpentier, J.-R. Falleri, F. Morandat, E. Ben Hadj Yahia, L. Réveillère, “Raters’ reliability in clone benchmarks construction,” Empirical Software Engineering, vol.22, no.1, pp.235–258, Feb. 2017.

[6] K.S. Jones, “A statistical interpretation of term specificity and its application in retrieval,” Journal of Documentation, vol.28, no.1, pp.11–21, 1972.

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

[8] T. Uno, “An efficient algorithm for solving pseudo clique enumeration problem,” Algorithmica, vol.56, no.1, pp.3–16, Jan. 2010.

[9] T.M.J. Fruchterman and E.M. Reingold, “Graph drawing by force-directed placement,” Softw., Pract. Exper., vol.21, pp.1129–1164, 1991.