Vol.2017-SE-197 No.7
2017/11/9

コードクローンに対するリファクタリング可能性に基づいた削減可能ソースコード量の調査

石津 卓也[1,a)]　吉田 則裕[2]　崔 恩瀞[3]　井上 克郎[1]

概要：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．また，集約とは，コードクローンを 1 つのメソッドやクラスなどにまとめることである．集約によりコードクローンの行数は削減される．集約で削減できるコードクローン行数を推定するとき、コードクローン片間での重複やリファクタリング可能性を考慮することで，より正確に測定できる．本稿では，クローンペア（コードクローンの関係にある 2 つのコード片）に対してリファクタリング支援ツール JDeodorant のリファクタリング可能性判定機能を用いて，オープンソースソフトウェアを対象とした削減可能ソースコード量の推定を行った．その結果，検出されたコードクローン行数のおよそ 6%が削減可能で，クローンセット数にすると全体の 19%ほどが集約対象になることが明らかになった．

キーワード：コードクローン，リファクタリング可能性，削減可能ソースコード量

Investigating the amount of reducible source code based on the refactorability of code clones

Takuya Ishizu[1,a)]　Norihiro Yoshida[2]　Eunjong Choi[3]　Katsuro Inoue[1]

1. まえがき

ソースコード行数が開発や修正によって増大するにつれてソフトウェアの保守は困難になる傾向にある．コード行数の増大の一因には開発者がソースコード作成を効率良く進めるためにコピーアンドペーストなど多用によって発生するコードクローンが挙げられる [4][7]．コードクローンとは，ソースコード中に存在する一致，または，類似した部分を持つコード断片のことである．コードクローンの削減はソースコード行数の削減に繋がると考えられるため，多くのコードクローンに関する既存研究が行われてきた [1][4][6][10][11]．コードクローンの削減に貢献する手法として集約と呼ばれる手法が存在する．集約は互いにコードクローンの集合 (以下，クローンセットと呼ぶ) に対して，プログラムの振る舞いを保ちながらメソッドやクラスを作成し元のコードクローンから呼び出すことによってコードクローンを削除する修正手法である．集約を行うにはコストや手間といったコストが必要になるため，開発者は集約を行う動機となる指標が必要となる，

本稿では，その指標として開発者を支援する削減可能ソースコード量を定義した．削減可能ソースコード量は，コードクローンに対して集約手法を適用してコードクローンを削除して減少する行数の推定値である．ただし，削減可能ソースコード量の算出には 2 つの解決すべき課題が存在する．1 つ目の課題は検出されたコードクローンの存在範囲が別の異なるコード片を持つコードクローンと重複するオーバーラップと呼ばれる現象が生じる点である．オーバーラップ下における複数のクローンセットに対する集約の実作業を緻密に行おうとするのは非常に労力がかかる恐

[1] 大阪大学
Osaka University
[2] 名古屋大学
Nagoya University
[3] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[a)] t-ishizu@ist.osaka-u.ac.jp

れがあり，本稿ではこの課題を解決するために，大まかな集約手法を採用するものとして，削減可能ソースコード量を算出することとした．なお，コードクローン検出ツールは存在範囲が部分的に異なる多様なコードクローンを検出するため，オーバーラップに関する問題を避けて削減可能ソースコード量を算出することはできない．2 つ目の課題は，コードクローンのリファクタリング可能性である．本稿では，この課題を解決するために，検出したクローンペア（コードクローン関係にある 2 つのコード片）に対して，リファクタリング支援ツール JDeodorant のリファクタリング可能性判定機能を利用して問題の解決を図った．本稿ではこれらの課題を解決することで，より正確な削減可能ソースコード量を算出する手法として提案する．

調査方法では，コンパイル可能であった 7 つの Java プロジェクトに対して削減可能ソースコード量の調査を行った．具体的に，まず，コードクローン検出ツール CCFinderX[9] を用いてコードクローンを検出する．次に，JDeodorant のリファクタリング可能性判定機能は検出されたクローンペアごとにリファクタリング可能性を評価する．リファクタリングが不可能なコードクローンを集約候補から除外する．さらにオーバーラップしているコードクローンに関してもメタヒューリスティック [3][12] を利用して除外する．メタヒューリスティクスとは，特定の問題に依存しない，組み合わせ最適化問題における近似解を求める解法をもつアルゴリズムの基本的な枠組みである．こうしたリファクタリング可能性とオーバーラップに関する 2 つのフィルタリングで除外されなかったコードクローンに対して削減可能ソースコード量を算出した．

2. 背景

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一致，または，類似した部分を持つコード断片のことである [4][7]．また，コードクローン関係にある二つのコード片をクローンペアと呼び，コードクローンの集合をクローンセットと呼ぶ．また，コードクローンには 3 つのタイプが存在する．タイプ 1 は空白行やコメント行などのコーディングスタイルを除いて完全に一致するコード片を持つ．タイプ 2 は変数名や関数名，変数の型などの識別子のみが異なるコードクローンである．そして，タイプ 3 はタイプ 2 である上に命令文の挿入や削除，変更が行われたコードクローンを指す．コードクローンの主要な発生原因にはソフトウェアの開発時や修正時に開発者が作業効率の向上を目的とした既存のコード片のコピーアンドペーストが挙げられる．コピーアンドペーストによって作成されたコードクローンは類似の機能を開発する際に作業効率の代償としてソースコード行数を増大させてしまう傾向がある．そして，ソースコード行数の増大はソフトウェア保守を困難にする．例

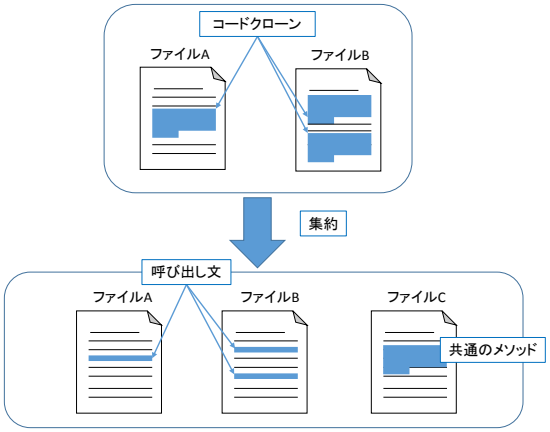

図 1 コードクローンとその集約例の図

えば，コード片に修正が加えられる場合，それとコードクローンになっているすべてのコード片に対して一貫性のある修正を検討する必要性を求める．これは修正の検討漏れが動作の不具合を発生させる可能性を秘めているためである．

以上の困難さを解決するには開発者はコードクローンの所在を逐一把握する必要がある．そのコードクローンの所在はコードクローン検出ツールによって検出可能である [5][8][14]．しかし，大規模なソフトウェア開発において常にコードクローンの発生を監視する必要があり，非常に煩雑な作業が伴うと予想される．そこでコードクローンを共通のメソッドやクラスにまとめて，コードクローンは呼び出し文に置換して削減する操作として集約という手法が考案されてきた．

図 1 は二つのファイル A，B から検出されたタイプ 1 あるいはタイプ 2 のコードクローンを集約した例を表している．コードクローンを図 1 では，ファイル C にまとめている．ただし，どこに共通のコード片をまとめるのか判断するには，各コードクローンが存在する階層構造や継承関係などを考慮する必要があり，必ずしも新しいファイルの作成を推奨しているわけではない．集約されたメソッドには，メソッドの導入文と終了文の 2 行を新たに加える必要がある．また，プログラムの振る舞いを保つためにコードクローンであったコード片を共通化したメソッドを呼び出す文に置換する．なお，タイプ 3 のコードクローンの集約はさらなる工夫が必要になるため，本研究では取り扱わず今後の課題としたい．最後に本稿では，これら以外の import 文やコメント文などのプログラムの本質に関わらないと想定される追加行数については考えないものとして議論する．

2.2 削減可能ソースコード量

コードクローン検出ツールを用いて検出されたコードク

ローンを集約する過程で，開発者は各コード片の所在や階層構造に着目をしてソフトウェア全体の振る舞いを変えないように慎重に作業を進める必要がある．このような作業に必要なコストや手間といった労力に対して見合う成果が得られるのか判断する指標が重要である．すなわち，開発者は指標を見て対価に見合うと判断すれば，ソフトウェアに存在する冗長なコード片を整理するための作業を積極的に行えると考えられる．本稿では，その指標として削減可能ソースコード量を定義している．削減可能ソースコード量とは，仮にコードクローンを集約した場合に，コードクローンの削除によって元のソースコードから減少するであろうソースコード行数の推定値を表す．具体的な計算手法は次章で説明する．

ソフトウェア全体から検出されたコードクローンの削減可能ソースコード量を求めるには，これから詳細に説明する 2 つの課題を解決する条件を満たしたすべてのクローンセットについて，数式 $T(S)$ の総和によって求められる．2 つの課題については次の 2 つである．

2.2.1コードクローンのオーバーラップ

2.2.2　コードクローンのリファクタリング可能性

詳細は次項以降で説明する．

2.2.1　コードクローンのオーバーラップ

1 つ目の課題はコードクローンのオーバーラップである．トークンに基づいたコードクローン検出ツールは，異なるトークン列ではあるものの存在範囲が重複しているコードクローンを複数検出する．異なるトークン列を持っているので別コードクローンとして扱えて，それぞれで削減可能ソースコード量を算出できるのだが，存在範囲が重複しているためオーバーラップしているクローンは同時に集約できない．また，クローンセットに属するコードクローンが別のクローンセットに属するコードクローンとオーバーラップしている時，それらのクローンセットはオーバーラップしていると表現する．

コードクローン c_1 と c_2 のオーバーラップ関係は次の数式で表現される．ただし，2 つのコードクローンは同一ファイル内に存在していて，どちらもクローン検出ツールなどで開始トークン番号と終了トークン番号が判明しているものとする．

$$(t_{end}(c_2) - t_{begin}(c_1))(t_{end}(c_1) - t_{begin}(c_2))$$

この数式が 0 以上であるならば，2 つのコードクローンはオーバーラップしている．負の数であるならば，2 つのコードクローンはオーバーラップしていない．$t_{end}(c)$ と $t_{begin}(c)$ はそれぞれコードクローン c の開始トークン番号と終了トークン番号を指している．この数式は 2 つのコードクローンの 6 通り存在する位置関係全てに対応している．

(1) c_1 の開始トークン番号が c_2 の開始トークン番号より前に存在して，c_2 とオーバーラップしない場合．

表 1　オーバーラップしているコードクローン例

行番号	1-1	2-1	2-2	3-1	プログラム文
1	+	+			if(bool1){
2	+	+			...
3	+	+			}
4	+		+		if(bool2){
5	+		+		...
6	+		+		}
7	+			+	while(roop){
8	+			+	...
9	+			+	}

(2) c_1 の開始トークン番号が c_2 の開始トークン番号より前に存在して，かつ c_1 の終了トークン番号が c_2 の終了トークン番号より前に存在していながら c_2 とオーバーラップする場合．

(3) c_1 の開始トークン番号が c_2 の開始トークン番号より前に存在して，かつ c_1 の終了トークン番号が c_2 の終了トークン番号より後ろに存在していながら c_2 とオーバーラップする場合（c_1 が c_2 を内包している）．

これらの位置関係に加えて，c_1 と c_2 を入れ替えれば 6 通りになる．

表 1 は，オーバーラップが発生するような代表的なソースコードを簡潔に表現したものに対して，3 つのクローンセットにそれぞれ属する四つのコードクローンについて対応する関係を示している．各クローンセットおよびコードクローンには互いに識別するために ID が割り振られており，例えば，1 – 1 はクローンセット ID1 のクローンセットの中でクローン ID（CID）1 に割り振られたコードクローンであることを示している．プログラムは 2 つの IF 文（1 行目から 3 行目と 4 行目から 6 行目）と一つの WHILE 文（7 行目から 9 行目）によって構成されている．クローン検出器は検出条件としてしきい値などを満たしているコードクローンであれば，表 1 のように 3 つのクローンセットに分けて検出する可能性がある．このとき，クローンセット ID1 と ID2，ID1 と ID3 はそれぞれオーバーラップしている．

2.2.2　コードクローンのリファクタリング可能性

2 つ目の課題はコードクローンのリファクタリング可能性である．コードクローン検出ツールが検出するコード片は条件分岐やクラス階層などは考慮されていない．例えば，集約されたメソッドの返り値は最大で 1 つの型しか持てないが，コードクローンによっては条件分岐によって必ずしも同じ型の返り値を持つとは限らない．あるいは，それぞれのクラスがディレクトリ構造で見たときに非常に離れていれば，その集約で考慮しなければならない制約条件は複雑になる．このような複雑な要因を抱え込まないコードクローンをリファクタリング可能であるとする．ただし，コードクローンのリファクタリング可能性はプログラ

ム言語の機能に大きく依存すると考えられる．そのため，リファクタリング可能性を考える際には対象とするプログラミング言語を事前に決定する必要がある．本稿では，前述にある通り Java 言語を対象としている．

Tsantalis らは Java 言語ソフトウェアを対象として検出されたコードクローンのリファクタリング可能性に関する研究 [13] を行っている．彼らは，Java 言語で記述されたソフトウェアから検出されたクローンペアを入力として，そのクローンペアのリファクタリング可能性について判断するための条件の洗い出しや手法が提案している．また，彼らは提案手法に基づいて Eclipse プラグイン JDeodorant を開発した．JDeodorant はリファクタリングを支援するツールの機能としてリファクタリング可能性の判断が可能である．そのツールは複数のコードクローンがリファクタリング可能になるのに必要な制約条件とネスト構造木，プログラム依存グラフを利用することによりリファクタリング可能性を評価していて，非常に高い精度で評価が可能になっている．本研究では，コードクローンのリファクタリング可能性の問題を解決するために JDeodorant のリファクタリング可能性判定機能を使用した．

3. 削減可能ソースコード量の算出方法

3.1 削減可能ソースコード量

ある 1 つのクローンセット S に属する n 個のコードクローン c_i を入力として得られる削減可能ソースコード量 $T(S)$ は次の数式 (1)〜(3) によって与えられる．

$$size(c) = l_{end}(c) - l_{begin}(c) + 1 \tag{1}$$

$$c^*_{size} = \frac{1}{n} \sum_{c_i \in S}^{n} size(c_i) \tag{2}$$

$$T(S) = n * c^*_{size} - (c^*_{size} + 2 + n) \tag{3}$$

数式 (1) について，$size(c)$ はコードクローン c が存在する行数を表していて，$l_{end}(c)$ と $l_{begin}(c)$ はそれぞれコードクローン c の終了行番号と開始行番号を示している．これらの情報はクローン検出ツールによって得られる．数式 (2) は，n 個のコードクローンの平均行数 c^*_{size} を示している．これは同じコードクローンであっても，開発者ごとに改行などの好みのコーディングスタイルが存在するために必ずしもすべてのコードクローンが同じ行数を持っていないと想定しているためである．そのため，平均的なコードクローン c^* を導入した．

数式 (3) で求められる削減可能ソースコード量はマイナス記号を境目にその前後で別々のソースコードの状態を計算している．マイナス記号より前では，元のソースコードに含まれるクローンセット S に属する n 個のコードクローン c_i 全体のソースコード量を表している．マイナス記号より後では，2 つの観点で計算がなされている．1 つ目は $c^*_{size} + 2$ の部分であり，ここでは抽出されたメソッドの行数を表している．+2 と加えられているのは，メソッドの導入文および終了文を考慮しているためである．なお，ここでの計算は主に Java 言語を対象としているが，ほかの言語では必ずしも 2 行の追加でサブルーチンの記述が完了するわけではないことに気を付けたい．

2 つ目の式は n であり，これはすべてのコードクローンのコード片を呼び出し文に置換した場合に必要な行数と想定している．ここでも，本質的には呼び出し文は 1 行で完結するものとしているが，他の言語では必ずしもそうではない．これら 2 つの計算部分がコードクローン削除によるソフトウェアの振る舞いの変更を防ぐための記述に必要な最低限の行数である．よって，元のコードクローンの行数とこれらの差が削除可能ソースコード量である．

3.2 コードクローンのオーバーラップ

オーバーラップの問題を解決するために，いくつかの手法が提案された．1 つ目はオーバーラップが存在しないコードクローン検出，例えば，メソッド単位のコードクローン検出を行うという案である．2 つ目に，Edwards III ら [2] によって提案された手法で，オーバーラップしているコード片間で，そのオーバーラップしているクローンセットの組み合わせに応じて新たなコードクローンへと分割していく手法である．2 つ目の手法では，表 1 はクローンセット ID2 と ID3 を集約することになる．最後に，オーバーラップしているクローンセットの中から，削減可能ソースコード量に基づいて集約すべきクローンセットの組み合わせを求める手法である．本稿では，3 つ目の集約すべきクローンセットの組み合わせを求める手法を採用した．これには，次の 2 つ理由があるから 3 つ目の手法が適していると判断したためである．

- コードクローン検出ツールに依存しない．
- 集約の手法はできる限り簡潔なものにする．

次に，集約すべきクローンセットの組み合わせを求める手法について説明する．まず，クローンセットの組み合わせを考えなければならない理由について説明する．クローンセットがオーバーラップしている時には一部のコード片が集約できなくなるため，集約するクローンセットの取捨選択をする必要がある．表 1 では，クローンセット ID1 だけを集約するのか，クローンセット ID2，ID3 を集約するのでは変換された後のソースコードに違いがある．削減可能ソースコード量の算出で説明した通り，呼び出し文の数や集約されたメソッドの追加行数に差異が生まれる．また，表 1 のプログラム文だけではクローンセット全体の削減可能ソースコード量が算出できない．すなわち，表 1 だけでは，クローンセット ID1 だけを集約する組み合わせを選んだ方が削減可能ソースコード量は大きくなるが，クローンセット ID2，ID3 を集約した方が，コード片の粒度がより

小さいために多くのコード片に適用可能なので，削減可能ソースコード量が大きくなる可能性が秘められていることに留意したい．これらが集約するクローンセットの組み合わせを考えなければならない理由である．

これらの組み合わせを素直に全通り計算する場合，k 個のオーバーラップが含まれる時に全通りの組み合わせを計算するのに必要な時間計算量は $O(2^k)$ である．実際はこれよりも計算量が低くなる傾向にあるが，それでも大規模なソフトウェアを対象にした場合に，非常に厳しい実行時間が求められることになる．

そこで本稿では，膨大な時間計算量を落としながらもある程度の精度を保つためにメタヒューリスティックな手法を採用した [3][12]．メタヒューリスティクスとは，特定の問題に依存しない，組み合わせ最適化問題における近似解を求める解法をもつアルゴリズムの基本的な枠組みである．我々の先行研究 [15] では，Java 言語のオープンソースソフトウェア（以下，OSS とする）に対してメタヒューリスティックを利用し，削減可能ソースコード量を推定を行っている．その研究においてメタヒューリスティックなアルゴリズムは 4 つ使用されている．提案されているアルゴリズムの中で，本研究では貪欲法の利用を採用した．これらの 4 つのアルゴリズムによる推定値の差はほぼないが，本研究では実行時間が特に短い貪欲法を採用した．

なお，貪欲法によるクローンセットの組み合わせを決定するアルゴリズムについて説明する．貪欲法の方針は各要素を評価値順にソートをして，評価値の高い順番に制約が許す限り解の候補に加える．本手法の場合，要素に相当するクローンセットにおける評価値および制約について説明する．評価値は削減可能ソースコード量である．削減可能ソースコード量が大きいほどそのクローンセットは集約後に削減できるソースコードの行数が多いということになる．また，制約がクローンセット間のオーバーラップになる．1 つのクローンセットを解の候補に加えれば，そのクローンセットとオーバーラップしているクローンセットは除外される．

最後にオーバーラップにおける考慮をしなければならない現象がある．クローンセットのオーバーラップには，1 つのクローンセット内で生じるものもある．この現象は SWITCH 文など同じ命令文が連続しやすいコード片で生じやすい．同じ命令文が続くコード片であるため，コード片を分割すれば集約できそうではあるが，本研究ではこのような現象が起きるクローンセット自体は除外するものとする．これは削減可能ソースコード量の算出手法に適用できない高度な集約である可能が高いためである．

3.3 コードクローンのリファクタリング可能性

Tsantalis らはリファクタリング可能なクローンペアにいくつかの制約を課す形でリファクタリング可能性を評価している [13]．その代表的な制約を以下の通りである．

(1) 変数のパラメータ化の際に制御依存，データ依存，反復操作や出力の振る舞いに変更があってはならない．
(2) それぞれ異なる子クラス型をもつ変数は，共通の親クラスで宣言されているか，あるいはオーバーライドされたメソッド内でのみ呼び出されている必要がある．
(3) フィールド変数のパラメータ化は，その値が不変の時のみ可能である．
(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) 抽出されたメソッドは，2 つ以上の変数を返していはならない．
(6) 条件付き return 文がコードクローンのコード片に含まれてはならない．
(7) 分岐処理を意味する命令文（BREAK，CONTINUE）があれば，それに対応する反復命令文がコード片に含まれていなければならない．

本研究ではタイプ 1 とタイプ 2 のコードクローンを集約対象としているために，タイプ 3 のコードクローンについて扱っている制約の記載は省略した．JDeodorant はクローンペアに対してネスト構造木やプログラム依存グラフを利用してクローンペアのコード片間の対応関係を調べて差異ある変数などをパラメータ化することで新たなメソッドに共通のコード片を抽出できるようにする．

JDeodorant のリファクタリング可能性判定機能を実行するには，Java 言語で記述されたソースコードに対する特定のコードクローン検出ツールの検出結果が必要になる．ただし，対象のプロジェクトは Eclipse 上でコンパイル可能である必要がある．JDeodorant のリファクタリング可能性の評価結果はクローンペア単位で出力される．すなわち，同じクローンセットに属しているコードクローン間において，リファクタリング可能なクローンペアが存在すればリファクタリング不能なクローンペアも存在している可能性がある．そこで，本研究ではクローンセットを 3 パターンに分類した．3 パターンについては次節で説明する．

3.4 算出手法

本研究で用いた削減可能ソースコード量の算出手法について説明する．以下にプロジェクトごとの削減可能ソースコード量を算出する 5 つの手順を示す．

Step1 **Eclipse 上に対象プロジェクトをコンパイルする．**
Step1 では，JDeodorant の利用のために Java プロジェクトをコンパイルする．JDeodorant は Eclipse 依存が強いプラグインとして開発されたツールである．そのため，Eclipse 上でプロジェクトをコンパイルできる必要がある．

Step2 **コードクローン検出ツール CCFinderX によってコードクローンを検出する．**

Step2 では，トークンに基づいたコードクローン検出ツール CCFinderX を利用する．JDeodorant はいくつかのコードクローン検出器のサポートしている．本研究では，CCFinderX を用いてコードクローンを検出する．

Step3　**コードクローン検出ツールの出力結果を入力として，JDeodorant による各コードクローンのリファクタリング可能性を評価する．**

Step3 では，Step2 で得た CCFinderX の出力結果を入力として JDeodorant を実行する．今回の調査においてはコンパイルエラーが出たプロジェクトは除外する．リファクタリング可能性はクローンペア単位でリファクタリング可能，リファクタリング不能，そして，階層構造や複数のメソッドをまたいでいるようなコード片を持つコードクローンに対しては評価そのものをスキップする明らかなリファクタリング不能の 3 種類に分けられる．リファクタリング可能とリファクタリング不能であったクローンペアが属するクローンセットは次の 3 パターンに分けられる．

パターン 1
: クローンセットに属するすべてのコードクローンに対して，どのクローンペアにおいてもリファクタリング可能である．

パターン 2
: クローンセットに属する任意のクローンペアにおいて，少なくとも 1 つ以上がリファクタリング可能で，少なくとも 1 つ以上リファクタリング不能である．

パターン 3
: クローンセットに属するすべてのコードクローンに対して，どのクローンペアにおいてもリファクタリング不能である．

パターン 1 とパターン 2 のクローンセットにはリファクタリング可能なクローンペアが含まれている．また，パターン 1 においてはクローンセットに対する削減可能ソースコード量の算出手法で求められる．パターン 2 のクローンセットにおいては，リファクタリング不能なコードクローンは除外する．

Step4　**リファクタリング不能なコードクローンを除外して，残るリファクタリング可能なコードクローンのオーバーラップ関係を抽出する．**

Step4 ではリファクタリング可能なクローンペア以外（パターン 2 の一部とパターン 3）は除外する．その後，リファクタリング可能なコードクローンに対してオーバーラップを抽出する．

Step5　**各クローンセットの削減可能ソースコード量を算出する．その後，貪欲法を用いてプロジェクト全体の削減可能ソースコード量を得る．**

Step5 では，オーバーラップ関係を解決するために貪欲法を用いて解の候補を選択する．そして，選択した解の候補であるクローンセットの削減可能ソースコード量の和をプロジェクト全体の削減可能ソースコード量とする．

4. 調査実験

この章では，Java 言語で記述された OSS を対象とするコードクローンのリファクタリング可能性に基づいた削減可能ソースコード量に関する結果について説明する．削減可能ソースコード量の最終的な目標は，開発者が大規模なソフトウェアから検出したコードクローンの除去作業を実行するべきか判断する 1 つの指標として利用されることである．この調査ではそのような指標として利用されるために必要となる削減可能ソースコード量を調査する．

4.1 調査対象のオープンソースソフトウェア

本調査で対象とした OSS について説明する．すべてのプロジェクトが主に Java 言語で記述されている．

表 2 は，対象とした 7 つの OSS プロジェクトとその行数，コードクローンが占める行数を表している．本研究では，既存研究 [13] や [15] で対象となったプロジェクトの中から，Java 言語で記述されていて，コンパイルが可能だったプロジェクトを載せている．調査手順における Step1 でコンパイル可能だったプロジェクトのみ選択している．表 2 でわかるように最もコードクローンが検出されたプロジェクトは JFreeChart である．

表 3 は CCFinderX で検出したコードクローンとクローンセットの個数，クローンセット 1 つあたりに属するコードクローンの個数を示すコードクローンメトリクス POP をまとめた表である．POP はコードクローンの総和からクローンセットの個数の除算で求められる．POP 数が大きいほど削減可能ソースコード量は大きくなると推測している．表 4 は検出されたクローンペアの個数と Step3 における JDeodorant で評価したリファクタリング可能なクローンペアの個数をまとめた表である．およそ全体のクローンペアの内、10%から 30%程度のクローンペアがリファクタリング可能なクローンペアになる．これはクローン検出ツールが多様なコードクローンを見つけようとして，複数にメソッドにまたがるコード片を持つようなコードクローンや類似した命令文の繰り返しをするようなコードクローンが検出結果に多く含まれているためと思われる．

表 5 は各 OSS から検出されたクローンセットに関して，リファクタリング可能性に基づいてパターンごとの個数をまとめたものである．すなわち，パターン 1 とパターン 2 がリファクタリング可能なクローンペアが含まれるクローンセットになる．パターン 2 は 3 つのパターンの中で最も低い割合を示した．部分的にリファクタリング可能になるパターンはディレクリ構造上機能は全く違うファイルに

表 2　調査対象の OSS(*単位はすべて $KLoC$)

プロジェクト名	バージョン	行数*	コードクローン行数*(%)
Apache Ant	1.10.1	268	60(22.3)
Columba	1.4	54	4.6(8.5)
JMeter	3.2	91	5.6(6.1)
JEdit	5.4.0	180	1.8(1.0)
JFreeChart	1.0.19	310	175(56.4)
JRuby	1.7.27	325	61(18.8)
Apache Xerces	2.10.0	238	83(34.9)

表 3　各 OSS から検出されたコードクローンとクローンセット

プロジェクト名	コードクローン数	クローンセット数	POP
Apache Ant	2981	264	2.96
Columba	335	135	2.48
JMete	437	201	2.17
JEdit	124	54	2.29
JFreeChart	9976	2309	4.32
JRuby	4383	1398	3.13
Apache Xerces	4889	1311	3.72

表 4　検出されたクローンペアとそのリファクタリング可能性

プロジェクト名	クローンペア数	リファクタリング可能（%）
Apache Ant	8785	2068(23.5)
Columba	597	187(31.3)
JMeter	294	99(33.7)
JEdit	100	17(17.0)
JFreeChart	84551	8765(10.4)
JRuby	15546	830(5.3)
Apache Xerces	44764	1612(3.6)

表 5　クローンセットのパターン分類　() 内の単位は%

プロジェクト名	パターン 1	パターン 2	パターン 3
Apache Ant	303(30.1)	50(5.0)	652(64.9)
Columba	37(27.4)	4(3.0)	94(69.6)
JMeter	43(21.4)	2(1.0)	156(77.6)
JEdit	14(25.9)	1(1.9)	39(72.2)
JFreeChart	449(19.4)	180(7.8)	1680(72.8)
JRuby	312(22.3)	39(2.8)	1047(74.9)
Apache Xerces	292(22.3)	82(6.3)	937(71.5)

記述されているが，振る舞いは同じであるようなコードクローンは共通のメソッドを作成するクラスを確保しにくいためにリファクタリング不能であるようなクローンペアが発生する．大規模なプロジェクトであるほど複数の機能を持っている可能性が高く，そのことにより JFreeChart や ApacheXerces の割合が高くなっていることがわかる．

4.2　調査結果

表 6 は各 OSS に対するリファクタリング可能行数とオーバーラップを考慮した削減可能ソースコード量が表になっている．括弧内の数値はコードクローン行数（表 2）に対する削減可能ソースコード量の比率となる．最小の削減率となったのは JRuby の 3.2%で，最大は Columba の 10.7%であった．全体ではおよそ 6%ほどの削減が見込めることが明らかになった．また，リファクタリング可能な行数に限ればおよそ 30%の行数を削減できることも分かった．表 7 は各 OSS に対すりリファクタリング可能クローンセット数とるオーバーラップを考慮した削減可能なクローンセット数になる．括弧内の数値は全クローンセット数（表 3）に対する削減可能クローンセット数の比率となる．これはリファクタリング可能性とオーバーラップの 2 つの課題によっておよそ 70%から 80%のクローンセットは除外されるということである．

図 2 は削減可能ソースコード量を算出過程において変化する除外されていないクローンセット数の棒グラフである．棒グラフの縦軸は最初に検出されたクローンセット数を 1 として変化したクローンセット数を正規化している．横軸は各 OSS について左から順番に最初に検出したクローンセット数，JDeodorant における評価でリファクタリング可能と評価されたクローンセット数，オーバーラップの考慮して貪欲法により削減可能とされたクローンセット数を表している．1 番目のフィルタリングにおける効果が強く，70%以上のコードクローンが除外されているのがわかる．2 番目のフィルタリングは検出しているコードクローンの絶対数の大きな JFreeChart や Apache Xerces などがやや減少しているのがわかる．

5.　まとめと今後の課題

本稿では，ソフトウェア開発者がコードクローンの削減によるソフトウェア保守を行うのを支援する指標の 1 つとして削減可能ソースコード量を定義した．そして，削減可能ソースコード量を算出する上で生じる 2 つの課題であるコードクローン間のオーバーラップとコードクローンのリファクタリング可能性について解決を図った．オーバーラップはクローンセット単位で削減可能ソースコード量がより大きいクローンセットを削減することにした．また，リファクタリング可能性はリファクタリング支援ツール JDeodorant のリファクタリング可能性判定機能を利用した．

調査実験では七つの Java 言語で記述された OSS を対象とした削減可能ソースコード量を測定して，その傾向を考察した．削減可能なソースコード量は検出されたコードクローン行数全体の 10%以下である．また，検出されたクローンセット数の 20%ほどが削減可能であった．

削減可能ソースコード量の算出結果からいくつかの課題が浮き彫りになる．一つは，実際に手動でソースコードからコードクローンを削減したときにどの程度の行数を削減

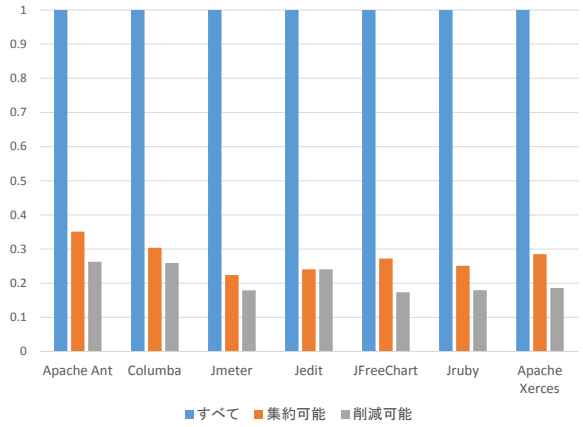

図 2 削減可能ソースコード量の算出過程におけるクローンセット数の変化

表 6 各 OSS のリファクタリング可能行数と削減可能ソースコード量 () 内の単位は%

プロジェクト名	リファクタリング可能	削減可能
Apache Ant	11224	3429(5.7)
Columba	1394	584(10.7)
JMeter	1117	385(6.1)
JEdit	384	136(6.6)
JFreeChart	30495	9700(5.0)
JRuby	7708	2161(3.2)
Apache Xerces	16611	5533(5.8)

表 7 各 OSS の削減可能なクローンセット (CS) 数 () 内の単位は%

プロジェクト名	リファクタリング可能 CS	削減可能 CS
Apache Ant	353	264(5.6)
Columba	41	35(25.9)
JMeter	45	36(17.9)
JEdit	13	13(24.1)
JFreeChart	629	401(17.4)
JRuby	351	251(18.0)
Apache Xerces	374	244(18.6)

できるのか議論しなければならない．ソフトウェア保守を支援する指標としての妥当性を得るために必要な検証と考える．また，対象が OSS に絞っている点について気をつけたい．大規模な商用プログラムでは，調査実験で得られたような結果にならない恐れがある．それは，検出するコードクローンの数やリファクタリング可能性に関する点で想定される．更に，プログラミング言語の違いも考えられる．削減可能ソースコード量の算出にはプログラミング言語の特性に大きく依存している．そのため，たとえば商用プログラムで使用される COBOL などの言語では今回得られた調査実験の結果とは乖離する可能性が考えられる．

謝辞 本研究は JSPS 科研費 25220003, 16K16034, 15H06344 の助成を受けたものです．

参考文献

[1] Bouktif, S., Giuliano, A., Merlo, E. and Neteler, M.: A novel approach to optimize clone refactoring activity, *Proceedings of the 8th annual conference on Genetic and evolutionary computation Proc. of GECCO 2006*, pp. 1885–1892 (2006).

[2] Edwards III, B., Wu, Y., Matsushita, M. and Inoue, K.: Estimating Code Size After a Complete Code-Clone Merge, 情報処理学会研究報告，Vol. 2016-EMB-41, No. 3, pp. 1 – 8 (2016).

[3] Harman, M., Phil, M., Jerffeson, de Souza, T. and Yoo, S.: Search Based Software Engineering: Techniques, Taxonomy, Tutorial, *Empirical Software Engineering and Verification*, pp. 1–59 (2012).

[4] 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).

[5] 肥後芳樹，楠本真二，井上克郎：コードクローン分析ツール Gemini を用いたコードクローン分析手法，電子情報通信学会，Vol. 105, No. 228, pp. 37–42 (2005).

[6] 肥後芳樹，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，Vol. 28, No. 4, pp. 43–56 (2011).

[7] 井上克郎，神谷年洋，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，Vol. 18, No. 5, pp. 47–54 (2001).

[8] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proceedings of the 29th international conference on Software Engineering Proc. of ICSE 2007*, pp. 96–105 (2007).

[9] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).

[10] 神谷年洋，肥後芳樹，吉田則裕：コードクローン検出技術の展開，コンピュータソフトウェア，Vol. 28, No. 3, pp. 29–42 (2011).

[11] Kim, H., Jung, Y., Kim, S. and Yi, K.: MeCC: memory comparison-based clone detector, *Proceedings of the 33rd International Conference on Software Engineering Proc. of ICSE 2011*, pp. 301–310 (2011).

[12] O'Keeffe, M. and Cinneide, M. O.: Search-based refactoring: an empirical study, *Journal of Software Maintenance and Evolution: Research and Practice - Search Based Software Engineering [SBSE]*, Vol. 20, No. 5, pp. 345–364 (2008).

[13] Tsantalis, N., Mazinanian, D. and Krishnan, G. P.: Assessing the Refactorability of Software Clones, *IEEE Transactions on Software Engineering*, Vol. 41, No. 11, pp. 1055–1090 (2015).

[14] 山中裕樹，崔 恩瀞，吉田則裕，井上克郎：情報検索技術に基づく高速な関数クローン検出，情報処理学会論文誌，Vol. 55, No. 10, pp. 2245–2255 (2014).

[15] 石津卓也，吉田則裕，崔 恩瀞，井上克郎：メタヒューリスティクスを用いた集約可能コードクローン量の推定，情報処理学会研究報告，Vol. 2016-SE-193, No. 5, pp. 1 – 8 (2016).