修士学位論文

題目

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の分析

指導教員
井上 克郎 教授

報告者
石津　卓也

平成 30 年 2 月 7 日

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 ソフトウェア工学講座

平成 29 年度 修士学位論文

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の分析

石津　卓也

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．主な発生原因はコピーアンドペーストなどの，開発を素早く進めるための行動である傾向がある．冗長的なソースコードはソフトウェア保守の際に，開発者が意図しない潜在的なバグを発生させる要因になりうる．そのため，冗長的なソースコードを発生させやすい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 つと言われている．

冗長的なコード片であると認められたコードクローンを 1 つにまとめる目的で，リファクタリングと呼ばれる作業が行われることがある．リファクタリングとは，共通のメソッドやクラスなどを新規に作成して，コードクローン自体は共通のメソッドやクラスの呼び出し文に置換することである [14]．ただし，プログラム全体での振る舞いはリファクタリングの前後で一貫している必要がある．このリファクタリングによってコードクローンが削除されるので，ソフトウェアから潜在的なバグの要因を軽減させることができると考えられている [1]．

企業に委託される大規模なレガシーコードのリファクタリングなどでは，冗長なソースコードを削除する目的の一環としてコードクローンのリファクタリングが行われることがある．あるいは，そのようなソースコードから検出されたコードクローンをリファクタリングする際に，その作業に必要なコストや効果を見積もる必要が求められるかもしれない．しかしながら，ソースコードの規模が大きくなるほど，コードクローンのリファクタリングは容易でなくなるため，その見積もりも難しいと考えられる．あるいは，これは検出されるコードクローンの数が増加したり，それらが検出される範囲が広くなったりすることが原因で付随する問題が発生するからである．

本研究では，コードクローンの増加に関して付随した問題を取り除き，コードクローンによってもたらされた冗長的な削減可能ソースコード量（本研究ではリファクタリング作業を通じて増減するソースコードの行数という意味で用いる）を推定する手法を議論する．コー

ドクローンの増加に付随する問題について，主に 2 つの問題を取り上げる．1 つは，コードクローンのリファクタリング可能性である．もう 1 つは，コードクローンの所在が検出時に部分的に重なることで生じるオーバーラップである．

また，我々はこの手法を議論する上で，7 つのオープンソースソフトウェア (OSS) の削減可能ソースコード量を推定している．その結果として，各 OSS の削減可能ソースコード量は検出されたコードクローンのおよそ 6%に及ぶことが分かった．更に，我々はこれらの OSS のコードクローンに対して，9 つのクローンメトリクスの傾向を分析した．これは，検出されたコードクローンのメトリクスから削減可能なコードクローンの傾向を得るために行った．その結果として，8 つのクローンメトリクスに関して，削減可能なコードクローンに有意差が認めれることが分かった．

主な用語

コードクローンのリファクタリング
削減可能ソースコード量
リファクタリング可能性
オープンソースソフトウェア

目 次

1　はじめに

ソフトウェア保守の問題の 1 つとして，冗長的なソースコードの増大が挙げられる．ソフトウェア開発では，類似の機能を複数実装する際に，開発者はコピーアンドペーストなどを用いて，実装の完成を優先した開発をしばしば行う．冗長的なソースコードが紛れ込むことは，ソフトウェア保守の際に開発者が意図しない潜在的なバグを生み出す要因となってしまう．

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 [6, 9]．主な発生原因はコピーアンドペーストなどの，開発を素早く進めるための行動である傾向がある．冗長的なソースコードを発生させやすい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 つと言われている．例えば，保守作業の一環としてソースコードの一部を仕様変更のために書き換えたとする．この時，書き換えたソースコードがコードクローンの一部であった場合，類似の変更を他のソースコードでも検討する必要がある．それは，類似の実装が他のソースコードでも行われている可能性があり，検討なしに放置した際に意図しないバグが紛れ込むことになってしまうためである．よって，開発者がコードクローンの管理や修正を行うことは，潜在的なバグの発生を軽減することに繋がるため，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 [2][6][8][12][13]

本研究では，リファクタリングと呼ばれる冗長的なコード片であると認められたコードクローンを 1 つにまとめる作業に注目する．リファクタリングとは，共通のメソッドやクラスなどを新規に作成して，コードクローン自体は共通のメソッドやクラスの呼び出し文に置換することである．ただし，プログラム全体での振る舞いはリファクタリングの前後で一貫している必要がある．リファクタリングは，最終的にコードクローンを削除することになるので，ソフトウェアから潜在的なバグの要因を軽減させることができると考えられている．

次に本研究の動機について説明する．企業に委託される大規模なレガシーコードのリファクタリングなどでは，冗長なソースコードを削除する目的の一環としてコードクローンのリファクタリングが行われることがある．あるいは，そのようなソースコードから検出されたコードクローンをリファクタリングする際に，その作業に必要なコストや効果を見積もる必要が求められるかもしれない．例えば，経費や期間におけて小さな労力であるにもかかわらず，コードクローンをリファクタリングすることで大きな恩恵が得られると判断できるならば，開発者にとってリファクタリングを行う契機となりうる．しかしながら，ソースコードの規模が大きくなるほど，複数のクラス間に跨るようなコードクローンのリファクタリングは容易でなくなるため，その見積もりも難しいと考えられる．あるいは，これは検出されるコードクローンの数が増加したり，それらが検出される範囲が広くなったりすることが原因で付随する問題が発生するからである．

本研究では，コードクローンの増加に関して付随した問題を取り除き，コードクローンによってもたらされた冗長的な削減可能ソースコード量（本研究ではリファクタリング作業を通じて増減するソースコードの行数という意味で用いる）を推定する手法を議論する．コードクローンの増加に付随する問題について，主に 2 つの問題を取り上げる．1 つは，コードクローンのリファクタリング可能性である．検出されたばかりのコードクローンに対して，リファクタリングが可能であるのかどうかの判断は困難であると考えられる．そこで，我々はコードクローンのリファクタリング可能性を判断するために JDeodorant という Eclipse プラグインを利用した．JDeororant はその機能の 1 つにクローンペア（コードクローンの関係にある 2 つのコード片）からリファクタリング可能性をバイナリ的に評価する．もう 1 つの問題は，コードクローンの所在が検出時に部分的に重なることで生じるオーバーラップである．我々はオーバーラップに対して，メタヒューリスティックによる近似アルゴリズム [5][15] を適用し，リファクタリングするべきコードクローンの組み合わせを選択的に決定する．本論文では，それぞれの背景と手法について，2 章と 3 章で説明する．

また，我々はこの手法を議論する上で，7 つのオープンソースソフトウェア (OSS) の削減可能ソースコード量を推定している．更に，我々はこれらの OSS のコードクローンに対して，9 つのクローンメトリクスの傾向を分析した．これは，検出されたコードクローンのメトリクスから削減可能なコードクローンの傾向を得るために行った．本論文では 4 章でこれらの結果を報告する．その後，5 章で全体のまとめと今後の課題について議論する．

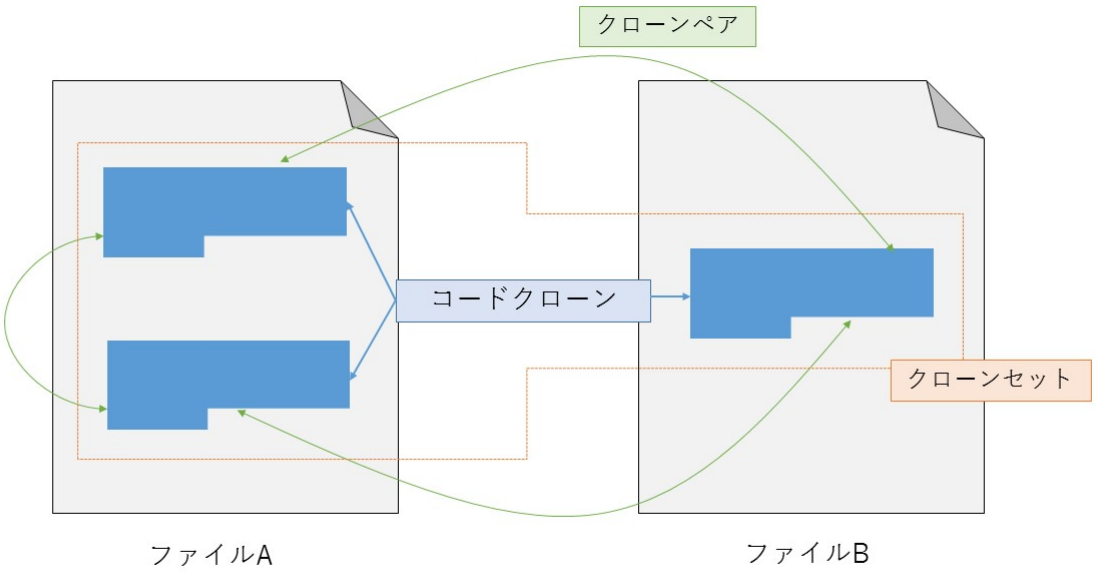

図 1: コードクローン

2　研究背景

本章では，コードクローンにまつわる用語の説明を行う．

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 [6, 9]．また，コードクローン関係にある 2 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，コードクローンは次の 3 つのタイプ分類できる．図 1 は，ファイル A とファイル B に存在するコードクローンを表している．図 1 で示す通り，コードクローンは複数のファイル間でも検出することがある．

- タイプ 1　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　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．

- タイプ3　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に開発時間を短縮する目的で既存のコード片をコピーアンドペーストするなどの作業が挙げられる．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時間短縮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．そのため，冗長性をもたらすコードクローンはソフトウェア保守において対処が求められる場合が多い．

すべてのコードクローンの所在を開発者が把握している場合は，ソフトウェア保守時に大きな問題は発生しない．しかし，ソフトウェアの規模が大きくなるにつれて，開発者の意図しないコードクローンんが発生したり，複数の開発者で開発したりするため，すべてのコードクローンの所在を把握するのは難しくなる．したがって，コードクローンの所在は一般的には自動で検出するツールを用いることが多い．

2.2　コードクローン検出ツール

コードクローンの自動検出手法として，プログラムの字句解析による行単位の検出や字句単位の検出，特徴メトリクスを用いた検出などがある [19, 10, 3, 16]．行単位の検出ではハッシュ関数を用いて，プログラムテキストの各行をハッシュ値に変換して，そのハッシュ値の列を対象として類似したハッシュ値列を求めることにより，コードクローンを発見する手法である．字句単位の検出では，閾値以上連続して一致する字句の部分列がコードクローンとして検出される．行単位で検出するよりも検出粒度が細かく，コーディングスタイルに依存しないなどの特徴を持つ．特徴メトリクスを用いたコードクローン検出では，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，その値の一致または近似の度合を調べるすることで，モジュール単位でのコードクローン検出を可能にする．

字句解析ベースのコードクローン検出ツールとして，CCFinderX がある [11]．CCFinderX は高いスケーラビリティを有しており, 大規模なソフトウェアに対しても実用的な時間でコードクローン検出が可能である．また, 変数名や関数名といったユーザ定義名や，変数の型などの一部予約語の違いなどの表現上の差異があるコードクローンを検出することができるという特徴も備えている．CCFinderX は様々な大規模ソフトウェアに適用されている [7]．

表 1: クローンメトリクス一覧

クローンメトリクス	説明
LEN	クローンセット内に含まれるコード片のトークン数の平均値
POP	クローンセット内に含まれるコード片の個数
NIF	クローンセット内に含まれるコード片を持つファイルの個数
RAD	クローンセット内に含まれるコード片を持つファイルの共通の親ディレクトリまでの距離
RNR	クローンセット内に含まれるコード片の非繰り返し度
TKS	クローンセット内に含まれるコード片の持つトークンの種類数
LOOP	クローンセット内に含まれるコード片の持つ反復処理の数
COND	クローンセット内に含まれるコード片の持つ条件分岐の数
McCabe	LOOP と COND の和

クローンメトリクス

コードクローンを様座な角度から計測して定量化した指標をクローンメトリクスと呼ぶ．4 章では，削減可能なコードクローンと削減困難なコードクローンについて，それぞれのクローンメトリクスを計測して，2 種類のコードクローンの傾向を観測する．その動機は，クローンメトリクスの傾向から削減可能なコードクローンの特徴を調査することである．例えば，削減可能なコードクローンのあるメトリクスの傾向が非常に大きかった場合，その傾向が他のプログラミング言語においても同様の傾向になる可能性はあると考えている．すなわち，今後の課題を見据えるために，クローンメトリクスについても言及している．

ここでは，本論文で計測する 9 つのクローンメトリクスについて，簡単に説明する．表 1 は，CCFinderX が検出したコードクローンから計測可能なクローンメトリクスの種類の一覧を表す．

クローンメトリクス *LEN* は，各コードクローンのトークン数を表す．トークンとは，ソー

スコード解析時に用いられる，ソースコードを構成する最小の要素を指す．解析工程を含むツールによって構成する要素の粒度や種類は異なる．本研究では，行数に着目した調査を行っているが，一般的に *LEN* の値が大きいほど，コードクローンの占める行数は大きくなるものと前提としている．

クローンメトリクス *POP* は，1 つのクローンセットに属するコードクローンの個数を表す．少なくとも 1 つ以上のクローンペアを持つ必要があるために，最小値は 2 である．この値が大きいほど，類似のコード片が多いと解釈されるため，そのコード片は非常に多岐にわたって利用されている可能性がある．

クローンメトリクス *NIF* は，コードクローンが存在するファイルの個数を意味している．NIF の最小値は，すべてのコードクローンが同じファイル上にある場合で，1 である．NIF が大きいほど，コードクローンとなっている機能が多岐にわたっていることを意味している．

クローンメトリクス *RAD* は，1 つのクローンセットに属するすべてのコードクローンが存在するファイルについて，それらファイルの共通の親ディレクトリまでの距離の中で最大のものである．最小値は，すべて同じディレクトリ内にある場合で，0 である．共通の親ディレクトリまでの距離が遠くなると，その分だけ多くの機能に渡るコードクローンが存在していることになる．

クローンメトリクス *RNR* は，コード片の非繰り返し度を示している．主に調査する必要がないコードクローンのフィルタリングに用いられる [21]．*RNR* は次のように定義されている．f をクローンセット S に含まれるコード片とする．$TOC(f)$ はコード片 f を構成している字句の数，$TOC_{repeated}$ はコード片 f を構成している字句の中で，繰り返し要素の字句の数を表すとする．この時，$RNR(S)$ は次のように表される．

$$RNR(S) = 1 - \frac{\sum_{f \in S} TOC_{repeated}(f)}{\sum_{f \in S} TOC(f)}$$

繰り返し要素が多いほど，RNR の値は小さくなる．連続した変数やメソッド呼び出し文などがそれに該当する．

クローンメトリクス *TKS* は，コードクローンに含まれるトークンの種類数を表している．CCFinderX では，検出するコードクローンの *TKS* の閾値を設定できる．その値は，12 が最小値となっている．

クローンメトリクス *LOOP* はコードクローンに含まれる反復処理の数である．また，クローンメトリクス *COND* は，条件分岐の数である．これらの和をとったメトリクスが *McCabe* である．これはコードクローンの複雑度を計測している．いずれも最小値は 0 である．

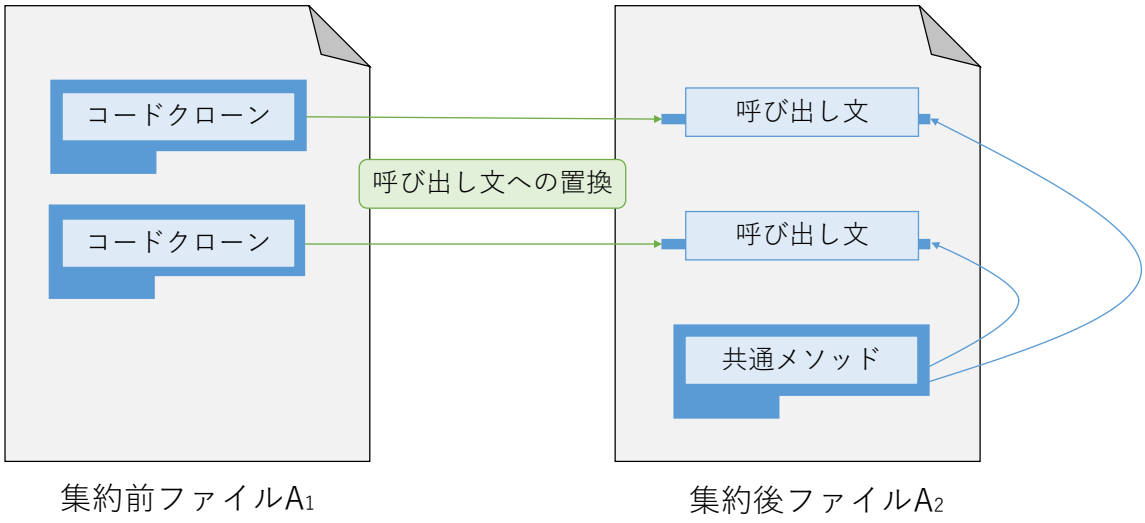

図 2: コードクローンのリファクタリング

2.3 コードクローンのリファクタリング

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．しかし，大規模なソフトウェア開発において定期的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リファクタリングという手法が考案されてきた．

図 2 はソースコードから検出されたコードクローンをリファクタリングした図を表している．コードクローンを図 2 では，新しいコード片として共通メソッドにまとめている．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る．そのため，必ずしも同一ファイル内に作成を推奨しているわけではない．

共通のメソッドには，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．ただし，これは，Java 言語における前提で，Java 言語以外のプログラミング言語ではこの限りではない．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 リファク

タリング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．

コードクローンのリファクタリングを導入することにより，ソフトウェア保守時における同時修正の負担が軽減される可能性がある．しかし，すべてのコードクローンがリファクタリングの実行が可能であるわけではない．これについては後ほど議論したい．

本研究では，次章以降に説明する手法の都合上，ソフトウェアの対象とするプログラミング言語を Java 言語とした．そして，Java 言語以外のプログラミング言語に関しては今後の課題としたい．研究対象とするプログラミング言語の数を拡張した場合に考慮すべきは，主にリファクタリング時の手法である．大まかな方針として，共通のサブルーチンの作成と呼び出し文への置換には変化はないが，プログラミング言語によってはこれらに制約が付加される可能性がある．

2.4 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リファクタリングする過程で，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費用や期間に対して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相当すると判断すれば，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めた．削減可能ソースコード量とは，仮にコードクローンをリファクタリングした場合に，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と予想される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．

ここでは，削減可能ソースコード量を算出する対象について議論したい．開発者のリファクタリング支援を目的とした場合，対象の候補としてはコードクローンの個数やトークン長，行数などが挙げられる．また，行数についてもどのような行数を対象とするのかで削減可能ソースコード量の意味が大きく変わる可能性がある．そのため，どのような量を対象とすれば，リファクタリングの前後のソースコードの変化をより効果的に表現できるのか考える必要がある．

本研究では，リファクタリングの前後で増減する行数に注目している．これは，リファクタリングの恩恵はソースコードの減少にあると考えているからである．例えば，メンバ変数に加工をするだけなどの非常にシンプルな行数の短いメソッドに含まれたコードクローンをリファクタリングした場合，そのクローンセットに属するクローンの個数が少ないほどリファクタリングによるソースコードの減少は少ない，あるいは増加する可能性がある．そし

て，シンプルなコードクローンは，シンプルではないコードクローンに比べて潜在的なバグが潜む可能性が小さいと考えた．そのため，我々は，リファクタリング後に減少する行数が大きいほど冗長性の高いコードクローンとみなし，リファクタリングの前後で減少する行数を削減可能ソースコード量を算出する対象とした．

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 つの問題を取り払った条件を満たすクローンセットを求める必要がある．2 つの問題については次の 2 つで，以降で説明する．

- リファクタリング可能性
- オーバーラップ

また，2 つの問題を取り払ったコードクローンを削減可能なコードクローンと呼び，また，JDeodorant にリファクタリングが困難と評価されたコードクローンやオーバーラップしていてメタヒューリスティクスによる候補から漏れたコードクローンを削減困難なコードクローンと呼ぶこととする．

2.4.1 リファクタリング可能性の問題

1 つ目の問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，リファクタリング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，そのリファクタリング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．

リファクタリング可能性を判定する機能を持つツール

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 言語で記述された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を行っている [17, 18]．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前提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．JDeodorant はバイナリ的にコードクローンがリファクタリング可能であるかリファクタリング困難であるのかを評価する．

JDeodorant で用いられるリファクタリング可能性の前提条件

Tsantalis らはクローンペアのリファクタリング可能性を示す前提条件を示している．これらの前提条件に違反するコードクローンはリファクタリング困難とみなされる．

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．

JDeodorant はタイプ 1 からタイプ 3 に分類されるクローンペアのプログラム依存グラフを作成して，それらの差異を比較したり，グラフを検索したりすることでリファクタリング可能性を評価する．

2.4.2 オーバーラップの問題

2 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リファクタリングできない．そのため，もしオーバーラップしているクローンセットの削減可能ソースコード量を重複して数えてしまった場合，オーバーラップしているクローンセットの削減可能ソースコード量と実際の削減行数とは異なる可能性を与えてしまう．そのため，クローンセットがオーバーラップしている場合，リファクタリングするコード片を分割したり，一方だけをリファクタリングしたりするといった工夫が必要になってくる．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．

$$isOverlap(c_1, c_2) = (t_{end}(c_2) - t_{begin}(c_1))(t_{end}(c_1) - t_{begin}(c_2))$$

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) オーバーラップしない場合:

$$(t_b(c_1) < t_b(c_2) \wedge t_e(c_1) < t_b(c_2)) \vee$$
$$(t_b(c_2) < t_b(c_1) \wedge t_e(c_2) < t_b(c_1))$$

(2) オーバーラップする，かつ，包含関係にない場合:

$$(t_b(c_1) < t_b(c_2) \wedge t_e(c_1) > t_b(c_2) \wedge t_e(c_1) < t_e(c_2)) \vee$$
$$(t_b(c_2) < t_b(c_1) \wedge t_e(c_2) > t_b(c_1) \wedge t_e(c_2) < t_e(c_1))$$

(3) オーバーラップする，かつ，包含関係にある場合:

$$(t_b(c_1) < t_b(c_2) \wedge t_e(c_1) > t_e(c_2)) \vee$$
$$(t_b(c_2) < t_b(c_1) \wedge t_e(c_2) > t_e(c_1))$$

(1) はオーバーラップしないための論理式である．(2) と (3) はどちらもオーバーラップをする関係を示した論理式であるが，包含関係にあるか否かで場合分けをしている．これらの論理式を 1 つにまとめたものが $isOverlap$ の数式になる．

表 2 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 a，b，c にそれぞれ属する 4 つのコードクローン c_{a1}，c_{b1}，c_{b2}，

表 2: オーバーラップしている簡単なソースコードの例

行番号	c_{a1}	c_{b-1}	c_{b-2}	c_{c-1}	プログラム文
1	+	+			if(bool1){
2	+	+			...
3	+	+			}
4	+		+		if(bool2){
5	+		+		...
6	+		+		}
7	+			+	while(loop){
8	+			+	...
9	+			+	}

c_{c1} について対応する関係を示している．クローンセット a はコードクローン c_{a1} を持つ．クローンセット b はコードクローン c_{b1}，c_{b2} を持つ．クローンセット a はコードクローン c_{c1} を持つ．

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，c_{a1} はクローンセット IDa 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 1 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 2 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．

メタヒューリスティクス

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のことである．アルゴリズムとして，局所的な探索に基づくアルゴリズムや，大域的な，個体群に基づくアルゴリズムが挙げられる．局所的な探索に基づくアルゴリズムとは，現在の解から近傍を求めて，現在の解を近傍の解に更新する作業を繰り返すことで近似解を求めるアルゴリズムである．大域的な個体群に基づくアルゴリズムとは，解の候補を生物の個体群に見立て最適解を求めるアルゴリズムである．このアルゴリズムでは，個体が次の世代に生き残るのかを決定する関数を工夫して生物の生殖や自然淘汰，突然変異などを再現する．これを繰り返して残った個体が最適

解に近似する．

メタヒューリスティックの解法の方針

メタヒューリスティクスは Search Based Software Engineering (SBSE) に基づいて，近似的アルゴリズムの解法の枠組みが与えられる [5]．メタヒューリスティクスは，特定の問題に依存しない手法のみに焦点を当てた考え方であり，あらゆる問題に対して汎用的に対応することが求められる．そのためメタヒューリスティクスでは，与えられた問題ごとに Representation，Operators，Fitness Function を定義することが SBSE では推奨される．これらは以下のように定義される．

- Representation 与えられた問題を表現している遷移可能な状態.
- Operators　遷移可能な状態に対して行う実行可能な操作.
- Fitness Function　現状の解が与えられた問題に対する最適解との差を評価する関数.

例えば，1 から 9 までの数字がランダムに 1 列に並んでいたときに，それらの数字に対して隣り合っている数字を入れ替える操作だけが許されている条件で，数字を昇順にソートする問題を考える．この問題をメタヒューリスティックを用いて解を求めるとする．このとき，この問題は，Representation，Operators，Fitness Function を用いて，次のように定義される．

Representation

はじめに，Representation を R(k)(k は 0 以上の整数) とし，1 列に並んでいる数字がどの順番で並んでいるのかを表現する．k 回目の数字の入れ替えがあった後の，1 列に並ぶ数字の状態を R(k) とする．

$$\text{定義の例：} R(k) = \{2, 5, 6, 1, 8, 3, 9, 7, 4\}$$

Operators

隣り合っている数字を入れ替える操作だけが許されているので，次のように表現できる．R の状態において，左から i 番目と i+1 番目の数字を入れ替える操作を Opr(R,i) で表す．例では i=5 より左から 5 番目の数字と 6 番目の数字を入れ替える操作になる．

$$\text{定義の例：} \quad R(k+1) = Opr(R(k+1), 5) = \{2, 5, 6, 1, 3, 8, 9, 7, 4\}$$

Fitness Function

例のソート問題における，最終的な解を考えると，左から i 番目の数字は i である．よっ

て，i 番目の数字を現状の解からの差の絶対値の和で評価する事が考えられる．すなわち，1 番目から 9 番目の数字の評価の総和が 0 であるとき，ソートが終了したことになる．例における，k 回目の評価値より k+1 回目の評価値の方が最適解 0 に近い評価値をとっているので，k+1 回目の方が最適解に近いと判断できる．ただし，局所的な最大値 (最小値) をとる場合があり，必ずしも評価値に近づくことが最適解を得られる保証はない．

$$\text{定義の例}：F(R(m)) = \sum_{m=1}^{9} |m - a[m]| \ \ (a[m] \text{ は } m \text{ 番目の要素})$$

$$F(R(k)) = |1-2| + |2-5| + |3-6| + |4-1| + |5-8| + |6-3| + \cdots + |9-4| = 24,$$

$$F(R(k+1)) = |1-2| + |2-5| + |3-6| + |4-1| + |5-3| + |6-8| + \cdots + |9-4| = 23$$

貪欲法

貪欲法は，最適化問題に対して，解の候補からその解の評価値を求めて，評価値の高い順番に解の候補を選択することで最適解を求めるアルゴリズムである．最適化問題によっては最適解を得ることは難しいが，簡単に近似解を得ることができるアルゴリズムなので，最適化問題に対する近似解を求めるために適用されることが多い．

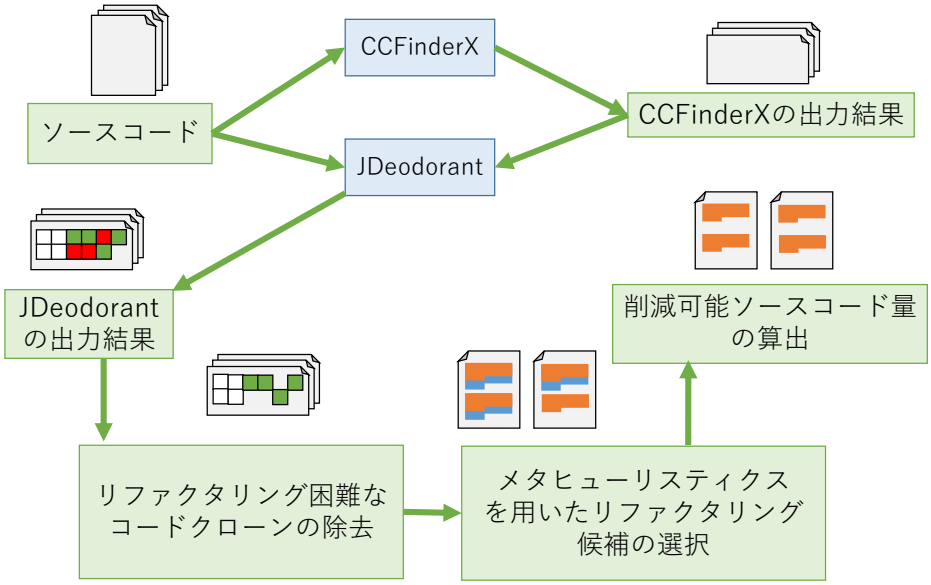

図 3: 削減可能ソースコード量算出の概要図

3 提案手法

この章では，削減可能ソースコード量を算出する手法を説明する．図 3 はその一連の流れを表している．削減可能ソースコード量を算出するためには，いくつかの工程を経る必要がある．

まず，初めに削減可能ソースコード量を算出する概要を説明する．

1. 対象とする Java プロジェクトを Eclipse 上でコンパイルする．これは，JDeodorant を実行するために必要な工程である．また，JDeodorant は Eclipse プラグインであるため，Eclipse 上でコンパイルするのが望ましい．

2. コードクローン検出ツール CCFinderX を用いて，プロジェクトからコードクローンを検出する．検出したコードクローンの所在情報は拡張子が ccfxd 形式で出力される．

3. JDeodorant に対して，ccfxd 形式のファイルとソースコードを入力することで，JDeodorant を実行する．JDeodorant の実行方法は後で説明する．また，実行が成功すると，リファクタリング可能性に関するクローンペアの分析結果がエクセル形式のファイルとして出力される．

4. JDeodorant の分析結果ファイルからリファクタリング困難なコードクローンを除外する．

5. リファクタリング困難なコードクローンを除外した分析結果ファイルに対して，残ったリファクタリング可能なコードクローンについて解析する．解析では，各クローンセットの削減可能ソースコード量とそれぞれのクローンセット間のオーバーラップを割出す．

6. オーバーラップしているクローンセットに対して，削減可能ソースコード量に基づいたメタヒューリスティクスを用いて優先してリファクタリングすべきクローンセットを探索する．

7. すべての削減可能なクローンセットの削減可能ソースコード量を総和することで，プロジェクト全体の削減可能ソースコード量を算出する．

1. は JDeodorant を実行する上で必要な手順となる．JDeodorant の環境構築は，Eclipse 上で行われるが，2 つのワークスペースを用意すると円滑に構築が進む．それは，JDeodorant は 2 つのワークスペース上にある同一のプロジェクトに対して解析とコンパイルテストを行っているためである．2. は CCFinderX の実行である．CCFinderX の実行方法は GUI による gemX と呼ばれる環境を用いる実行方法と，CCFinderX そのものに対して，コマンド ccfx による実行方法がある．どちらの方法を用いても，コードクローンを検出することは可能である．出力ファイルの ccfd 拡張子は，通常読むことはできないが，CCFinderX の P モードによる印字を行えば，txt 形式などにも変換可能である．3. 以降の説明は，次の節以降で詳細に説明する．3. と 4. は 3.1 で，5. と 6. については 3.2 で，7. については 3.3 で説明する．

3.1 リファクタリング可能性を評価する上での課題

ここでは，JDeodorant の出力結果，そしてその結果からリファクタリング困難なクローンペアを排除する手法について説明する．

JDeodorant の実行

JDeodorant にはコマンドラインで実行できる環境がある．リファクタリング可能性は，そのコマンドラインの ANALYZE_EXISTING モードによる出力結果で得られる．実行するためには，Eclipse の実行構成の中から，Eclipse アプリケーションを設定する必要がある．ワークスペースデータの設定やプロダクトの実行先は，適時適切なものを選択する．

コマンドライン引数には，対象となる Java プロジェクトのロケーション情報やエクセルファイルの入出力先，実行モードの指定，コードクローン検出ツールに関する情報が必要になる．特に，コードクローン検出ツールについて，JDeodorant は CCFinderX 以外のツールも指定できる．そのため，使用する検出ツールによって指定方法が異なるため，注意が必要である．

CCFinderX の場合，コードクローンの所在情報などをまとめた CCFinderX の検出結果 ccfxd ファイルと，各ソースコードの内容をトークンごとに解析したファイルのディレクトリ.ccfxprepdir の指定が必要になる．なお，.ccfxprepdir ディレクトリは CCFinderX 実行時に指定したディレクトリ直下に作成される．

JDeodorant の実行結果

JDeodorant の実行は 2 段階に分かれて行われる．

1 段階目では，ユーザ指定の xls ファイルに各コードクローンの所在情報として，クローンセット ID(エクセル内ではクローングループ ID となっている)，ソースフォルダー名，パッケージ名，クラス名，メソッド名，メソッドシグネチャ，開始行番号と終了行番号とそれらのオフセット値，クローンセットに属するコードクローンの数，クローンペア間の位置関係が解析される．クローンペア間の位置関係とは，同じ属するコードクローンの中で構造的に最大の距離にある関係を示している．例えば，同じファイル内にあっても異なるメソッド内，あるいは，異なるファイル内にクローンペアがあるといった表現がされる．

2 段階目では，更に 8 つのファイルが生成される．7 つの csv ファイルと 1 つの xls ファイルである．また，このほかにも多くの分析結果の HTML ファイルが生成される．

これらは形成したプログラム依存グラフを解析したもので，コードクローンのリファクタリング可能性についても確認できる．視覚的に理解しやすいのは，2 段階目で生成され，ユーザ指定のファイル名に-analyzed が追加された xls ファイルである．ファイルには 1 段階目で作成された xls ファイルに，リファクタリング可能性の情報が付加されている．図 4 の右部分にセルの色が緑や赤に塗りつぶされている範囲がそれである．上三角の形に塗られているのは，セルの 1 つずつがクローンペアのリファクタリング可能性を表しているからであり，すなわち，同じ組み合わせのクローンペアは最大で 1 組のみ表示しているためである．これらのセル情報に含まれる URL を開くと，ブラウザ画面から分析結果の HTML ファイルが開かれる．

赤色のセルはリファクタリング困難なクローンペアであり，定めている前提条件に違反するクローンペアである．反対に緑色のセルは，リファクタリング可能なクローンペアである．基本的には，この 2 色のどちらが指定されている．すなわち，リファクタリング可能性はバイナリ的に評価されることになる．ただし，色が塗られていないクローンペアが存在してい

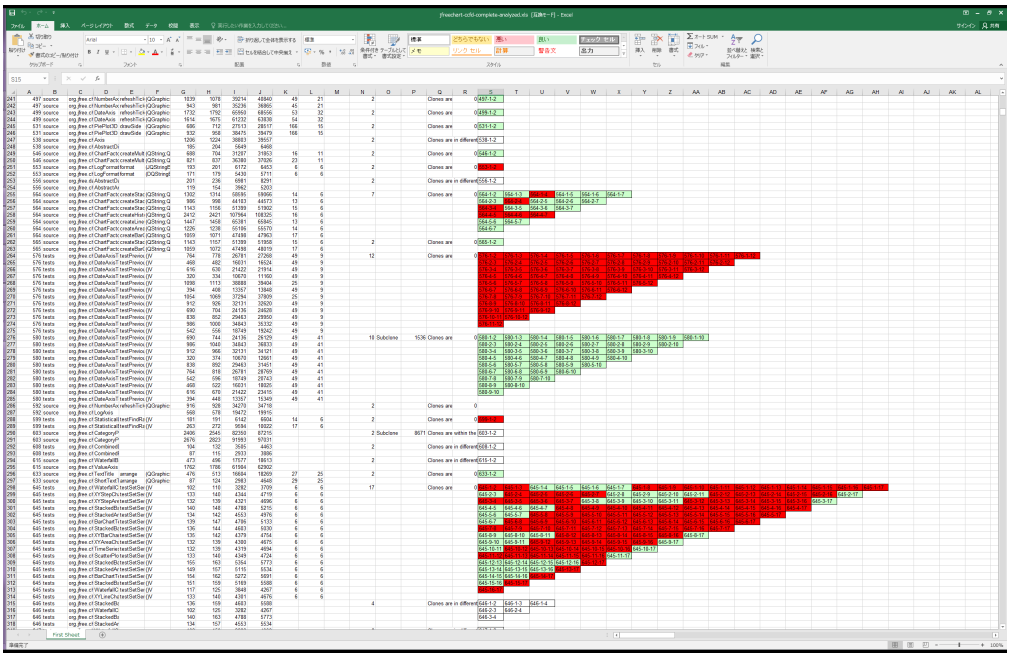

図 4: JDeodorant 出力ファイルの例

る場合，何らかの不具合かコードクローンに複数のメソッドが含まれていることが原因でリファクタリング可能性が正しく評価されていない．

あるいは，tree.csv ファイルでもリファクタリング可能性を確認できる．その中の RefactoringWasOK の項目に true または false でリファクタリング可能性がバイナリ的に評価されているのがわかる．

リファクタリング可能なクローンペアの抽出

リファクタリング困難なクローンペアは削減可能なコードクローンとはなりえないので，あらかじめ除外するのが好ましい．とりわけ，次の工程では，メタヒューリスティックを用いるので，候補となるクローンセット数を減らすと，計算時間量は減少することになる．

リファクタリング可能なクローンペアのみ抽出するためには，先の tree.csv を用いる．xls ファイルでも，クローンペアごとのリファクタリング可能性を色による判定で確認できるが，tee.csv ファイルの true または false による表現のほうがリファクタリング可能性を取得しやすいためである．ただし，リファクタリング可能性の評価に失敗しているクローンペアの情報は tree.csv から欠落しているので，実装の際には注意が必要である．

JDeodorant はクローンペア単位でのリファクタリング可能性を評価する．対して，コードクローンのリファクタリングはクローンセット単位で行うことを想定している．この溝を

埋めるために，同じクローンセット内にリファクタリング困難なクローンペアとリファクタリング可能なクローンペアが混在する場合を考える必要がある．

入力の tree.csv ファイルに対して，コードクローンを同じクローンセット ID をもつクローンセットごとに整理する．整理したあるクローンセット S の n 個コードクローンには，それぞれ対応する $n-1$ 個のクローンペアがあり，そのクローンペアに対して，リファクタリング可能性が評価されている．この時，各コードクローンのリファクタリング可能性は，次のように決定する．$n-1$ 組のクローンペアについて，順にリファクタリング可能性を見て，リファクタリング可能が 1 つでもあれば，そのコードクローンはリファクタリング可能とする．反対に，$n-1$ 組の中にリファクタリング可能なクローンペアがなければ，そのコードクローンはリファクタリング困難とする．クローンセット S 自体のリファクタリング可能性は次のようにした．

本研究では，クローンペアのリファクタリング可能性に基づいて，クローンセットを 3 種類に分類した．

- All クローンセットに属するすべてのコードクローンについて，どのクローンペアもリファクタリング可能であるクローンセット．
- Part リファクタリング可能なクローンペアとリファクタリング困難なクローンペアが混在しているクローンセット．
- Non クローンセットに属するすべてのコードクローンについて，どのクローンペアもリファクタリング困難であるクローンセット．

Part に分類されるクローンセットは，リファクタリング困難なコードクローンは除外されるので，削減可能コードクローン量の算出上では All に分類されるクローンセットと同じように扱う．しかし，Part に分類されるクローンセットは，一部のコードクローンが除外されているので，リファクタリング作業の観点からすると，その分だけ手間が増えることに留意したい．

3.2 オーバーラップの解決手法

この節では，主にオーバーラップしたコードクローンから削減するべきコードクローンを発見する手法について説明する．

オーバーラップを解決する方針の選択

まずはじめに，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．単純にコードクローンをリファクタリングしようにも，検出するツールによってはオーバー

ラップしているコードクローンの判別する手段すら困難であることがある．そのため，オーバーラップの問題を解決する手段は，開発者にとって容易である必要が求められる．

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．これは，コードクローン検出ツールの検出粒度が関連している．メソッドやクラスを対象とすると，オーバーラップが発生しない．メソッド単位のコードクローンの検出では，1 つのメソッド自体がコードクローンである必要がある．同時に他のメソッドと所在が重なることはないので，オーバーラップは起こりえない．クラス単位のコードクローン検出においても同様である．そのため，コードクローンの検出粒度を大きくすることはオーバーラップの発生を減少させることに繋がると考えられる．

2 つ目は，Edwards III ら [4] 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クローン間で，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．例えば，2 つのコードクローン A，B が部分的にオーバーラップしているとする．そのオーバーラップしている部分を新たなコードクローン C とみなす．コードクローン C となった部分以外だけで構成されるコードクローン A と B はそれぞれ A’ と B’ とみなす．そのため，リファクタリング時に作成する共通のメソッドは A’ と B’ と C から作られるメソッドの 3 つになる．すなわち，表 2 はクローンセット ID2 と ID3 をリファクタリングすることになる．3 つ以上のコードクローンがオーバーラップする場合も同様に考える．

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リファクタリング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 つ目のリファクタリング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．

- コードクローンを検出する粒度に依存しない．
- リファクタリングの手法はできる限り簡潔なものにする．

検出粒度の問題は，削減可能ソースコード量を算出する上で対応できるコードクローンのスケーラビリティが低下することがあげられる．また，開発者が削減可能ソースコード量を算出するための負担を少しでも減らすながらも，それなりの精度も求められる．

クローンセットの組み合わせを求める方針

次に，リファクタリングすべきクローンセットの組み合わせを求める手法の方針には考慮すべきことがある．

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リファクタリングできなくなるため，リファクタリングするクローンセットの取捨選択をする必要がある．表 2 では，クローンセット ID1 だけをリファクタリングするのか，クローンセット ID2，ID3 をリファクタリング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，呼び出し文の数やリファクタリングされたメソッドの追加行数に差異が生まれる．また，表 2 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 2 だけでは，クローンセット ID1 だけをリファクタリング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 をリファクタリングした方が，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．これらがリファクタリング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 [5][15]．我々の先行研究 [20] 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．

自身とのオーバーラップ

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リファクタリング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リファクタリングである可能が高いためである．

貪欲法の適用

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．評価値は次の節で説明する 1 つあたりのクローンセットの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リファクタリング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．

具体的なオーバーラップするクローンセット間に対する解決

図 5 はオーバーラップする 4 つのクローンセットへ貪欲法を適用する概要図を示している．各数字がクローンセット ID と呼ばれるもので，各クローンセットに属するコードクローンの所在情報から，クローンセット間のオーバーラップしているかどうかの状態を確認できる．図 3.2 では，クローンセット間で示される矢印がクローンセット間のオーバーラップを表現している．初期の状態では，4 つのクローンセットに，a，b，c，d が割り振られている．a は b と c とオーバーラップしていて，b は a と c とオーバーラップしている．また，d は a と c とオーバーラップしている．最後に c は b と d とオーバーラップしている他に c 自身ともオーバーラップしている．

自身のオーバーラップするクローンセットは必ずクローンセット単位でのリファクタリングができないので，事前に除外する．すると，3 つのクローンセット a，b，d が候補に残ることになる．a と b，a と d がオーバーラップしている．この中からリファクタリングすべきクローンセットを選択する必要がある．例えば，a は残る 2 つのクローンセットとオーバーラップしているので，a を候補に挙げると b，d は候補から外れる．反対に，b を選択すると a とオーバーラップしているために a は候補から外れる．d は b とオーバーラップしていないので候補に含める．そのため，候補は $\{a\}$ または $\{b,d\}$ の 2 通りであることがわかる．

これらの作業を不特定多数のコードクローンに適用するのは，困難であることは時間計算量 $O(2^k)$ から判断できる．そこで，部分的にオーバーラップしているクローンセット同士のみで 1 つの集合を形成して，それらをメタヒューリスティクスに適用する．図 5 に対して，貪欲法を用いた場合，各クローンセットの削減可能ソースコード量を計算する必要がある．この量をそれぞれのクローンセットの候補 $\{a\}$ または $\{b,d\}$ で計算する．仮に，$\{a\}$ の方が大きければ，$\{a\}$ が削減すべきクローンセットであるとした．反対に $\{b,d\}$ のほうが大きければ，$\{b,d\}$ を削減すべきクローンセットであるとした．

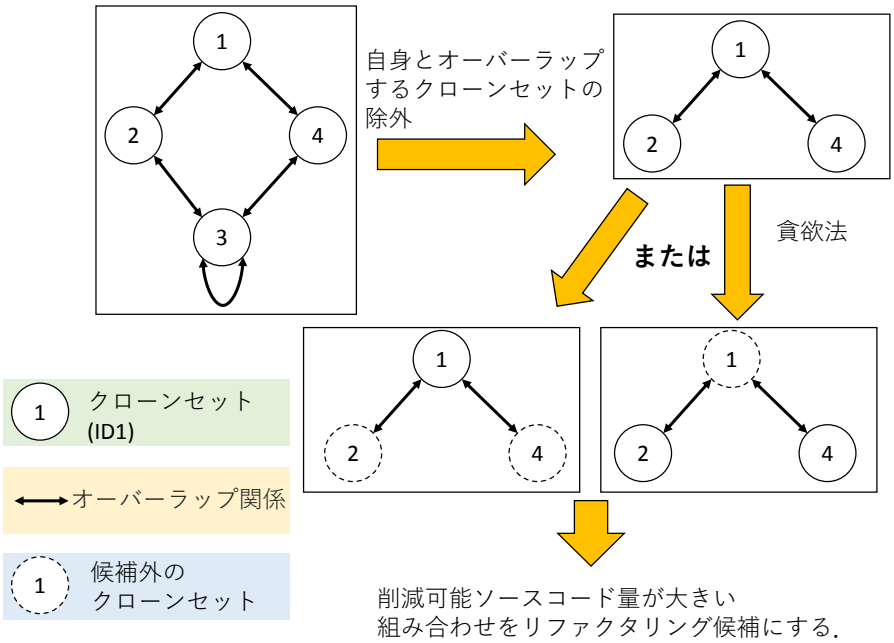

図 5: オーバーラップするクローンセットへの貪欲法の適用例

以上のようにして削減すべきクローンセットの集合をそれぞれのオーバーラップしているクローンセット間の集合に適用して求めることができる．そして，求めたクローンセットの候補の削減可能ソースコードを総和することでソフトウェア全体の削減可能ソースコード量を求めることができる．この手順は，貪欲法に限らず，そのほかのメタヒューリスティクスに分類される探索的アルゴリズムや遺伝的アルゴリズムでも適用ができる．しかし，貪欲法に限定すれば，すべてのクローンセットを対象としてそれぞれの削減可能ソースコード量を求めて，その削減可能ソースコード量が大きい順番に候補に追加しても同様のクローンセットの候補が得られる．なお，本手法および [20] の貪欲法は，後者の実装を行っている．これは実行時間にわずかではあるが短縮されることが挙げられる．オーバーラップしているクローンセットだけの集合の抽出する処理の分だけ実行時間が追加されるためである．

3.3 削減可能ソースコード量の計測手法

この節では，プロジェクト全体の削減可能ソースコード量を算出するために，クローンセット 1 つあたりの削減可能ソースコード量を算出手法を説明する．

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リファクタリングする過程で，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リファクタリングした場合に，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．

プロジェクト全体の削減可能ソースコード量を算出するには，オーバーラップも含みうるリファクタリング可能なクローンセットの削減可能ソースコード量をひとまず求めておいて，それらに基づいたメタヒューリスティクスで削減すべきクローンセットを割り出す．その後，削減すべきクローンセットの削減可能ソースコード量からプロジェクト全体の削減可能ソースコード量を算出する．

それでは，クローンセット 1 つあたりの削減可能ソースコード量の算出式について説明する．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．

4　適用実験

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．

4.2 章では，各オープンソースソフトウェアから検出されたコードクローンをリファクタリングした時の削減可能ソースコード量を算出した．これは，削減可能ソースコード量が実際のオープンソースソフトウェアに適用した際にどのような値を示すのかを調査するのが目的である．4.1 章はこの調査で利用するオープンソースソフトウェアについて，検出したコードクローンの特徴を踏まえて説明している．

4.3 章では，削減可能なコードクローンと削減困難なコードクローンの差異を調査する目的で行った．そのために，それぞれのコードクローンから計測できるクローンメトリクスを分析した．この性で得られた知見は今後のコードクローンの削減可能性の研究に貢献すると考えられる．

6 章では，JDeodorant によって削減困難と評価されたコードクローンについて，コード片を分割することによってリファクタリング可能性が変更するのかどうかを調査している．JDeodorant によるリファクタリング可能性は，できる限り安全で簡単なリファクタリングを開発者に提供するものとなっている．そのため，開発者がリファクタリング困難と評価されたコードクローンに対して，特定の操作を行うことでリファクタリングを行える可能性がある．個の可能性について，コード片を分割するという操作を投資て分析調査をしたい．

4.5 章では，削減可能ソースコード量の正当性を評価するために，JEdit から検出されて削減可能と評価したクローンセット 14 個に対して，手動でリファクタリングを行った．すなわち，削減可能ソースコード量は JDeodorant のリファクタリング可能性に大きく依存しているために，実際にはどの程度削減できるのかという評価が必要になる．この調査で得られた結果は JEit から検出されたクローンセットを対象としたものだが，他のオープンソースソフトウェアについても同様の傾向がみられる可能性が期待できるため，1 つのオープンソースソフトウェアを対象としている．

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 3 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 3: 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)

表している．本研究では，既存研究 [17] や [20] 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 1. 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．単位に示した LoC(Lines of code) はコード行数を表している．

表 3 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．ソースコード全体の 56.4%がコードクローンとなっていることがわかる．JFreeChart はグラフを作成するための Java のライブラリである．作成するグラフごとに機能を実装しているので，非常に多くのコードクローンが見られるものと考えられる．次に多いには Apache Xerces で 34.9%である．このプロジェクトは XML 文書のパースを操作するためのパッケージである．反対に，もっともコードクローン行数が少ないプロジェクトは JEdit である．

表 4.2 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．この POP はクローンメトリクスで，リファクタリング時に削除されるコードクローンの数がより多くなることが予想される．このメトリクスは後のクローンメトリクスの検定調査においても利用される．

コードクローン数が最も多いのは，JFreeChart で，コードクローン行数に反映されている．全体のおおよその傾向として，コードクローンの個数とコードクローン行数は比例することがわかる．また，POP についても同様の傾向が得られることが確認できる．

表 4: 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	2,981	264	2.96
Columba	335	135	2.48
JMeter	437	201	2.17
JEdit	124	54	2.29
JFreeChart	9,976	2,309	4.32
JRuby	4,383	1,398	3.13
Apache Xerces	4,889	1,311	3.72

表 5 は検出されたクローンペアの個数と 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．全体のクローンペアの内、最大で 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．最もリファクタリング可能なクローンペアが得られたプロジェクトは JFreeChart である．ただし，割合では，Columba や JMeter が 30%を超えているのが確認できる．最小は Apache Xerces で 3.6%であった．ここで，リファクタリング可能と評価されたクローンセットの個数が少ない理由は難しい．リファクタリング困難と評価されたクローンセットよりも評価そのものがされなかったクローンセットが多かったためである．我々はこの少なさを打開するために，1 つ対策をしたが，そのような場合でも効果を得られるプロジェクトとそうではないプロジェクトに分かれた．新しくした対策は後の節で説明する．

表 6 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，All と Part 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．

表 5: 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	8,785	2,068 (23.5)
Columba	597	187 (31.3)
JMeter	294	99 (33.7)
JEdit	100	17 (17.0)
JFreeChart	84,551	8,765 (10.4)
JRuby	15,546	830 (5.3)
Apache Xerces	44,764	1,612 (3.6)

表 6: 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	All	Part	Non
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)	1,680 (72.8)
JRuby	312 (22.3)	39 (2.8)	1,047 (74.9)
Apache Xerces	292 (22.3)	82 (6.3)	937 (71.5)

4.2　各オープンソースソフトウェアに含まれる削減可能ソースコード量の分析

表 7 は各 OSS の JDeodorant が示したリファクタリング可能なクローンセットの行数とそのクローンセット数である．また，表 8 は JDeodorant の結果を基に提案手法を用いた削減可能ソースコード量とそのクローンセット数を示している．ここでリファクタリング可能なクローンセットとは，*All* または *Part* に分類されるクローンセットのことである．また，提案手法における削減可能ソースコード量の算出に用いたクローンセットは，*All* または *Part* に属するクローンセットの中から，さらにオーバーラップをしていればメタヒューリスティクスを用いて候補に選ばれたクローンセットのことである．また，オーバーラップしていないクローンセットはそのまま候補となっている．削減可能ソースコード量の () 内の数値は，元のソースコードから検出された全コードクローン行数に対する比率を表してい

表 7: JDeodorant がリファクタリング可能と評価したクローンセット行数とクローンセット数 () 内の単位は%

プロジェクト名	JDeodorant 行数	クローンセット数
Apache Ant	11,224	353
Columba	1,394	41
JMeter	1,117	45
JEdit	384	13
JFreeChart	30,495	629
JRuby	7,708	351
Apache Xerces	16,611	374

る．JDeodorant がリファクタリング可能と評価したコードクローン行数のおよそ 3 分の 1 が削減可能にあることがわかる．これには理由が 2 つ存在すると考えている．1 つは，オーバーラップするコードクローンから優先してリファクタリングすべきコードクローンを選択したことがあげられる．このとき，コードクローンのオーバーラップの仕方次第では，削減可能な行数は減るはずである．2 つ目は，リファクタリングにおける振る舞いの変更を防ぐために書かれる共有メソッドと呼び出し文の存在である．例えば，*POP* が 2 の時であれば，削減可能ソースコード量はコードクローン行数の半分以下になることは想像できる．*POP* は表でもみた通り 2 または 3 であるので，この結果は予測しうる範囲内といえる．

元のソースコードから検出された全コード 9 クローン行数に対する比率は，およそ 6%にとどまることが判明した．OSS の場合，多くの開発者がメンテナンス作業に携わっているケースが想定されるため，この結果も予測しうる範囲内といえる．商業ソフトウェアを対象にした場合には，その冗長性にも依存するが，OSS とは異なる結果が得られる可能性があることに留意したい．

次に検出されるコードクローンの個数にも着目したい．JEdit の個数が変化していないのは，オーバーラップしているクローンセットが存在していないためである．JEdit 以外の OSS では，オーバーラップしているクローンセットが存在している．また，カッコ内の数値は，検出されたすべてのクローンセットの個数と比較した削減可能なクローンセットの個数の割合を示している．比率だけを見るなら，Apache Ant が最大の値 26.2%を示している．個数を見ると，最大は 401 個の JFreeChart である．JFreeChart の場合，比率は最小で 17.4%となっている．これは，多くのコードクローンを検出していても，その分だけ，オーバーラップしているコードクローンが多いことがあげられる．

表 8: 削減可能ソースコード量とクローンセット数 () 内の単位は%

プロジェクト名	削減可能ソースコード量	クローンセット数
Apache Ant4	3,429 (5.7)	264 (26.2)
Columba	584 (10.7)	35 (25.9)
JMeter	385 (6.1)	36 (17.9)
JEdit	136 (6.6)	13 (24.1)
JFreeChart	9,700 (5.0)	401 (17.4)
JRuby	2,161 (3.2)	251 (18.0)
Apache Xerces	5,533 (5.8)	244 (18.6)

4.3 クローンメトリクスの分析

動機

削減可能なコードクローンと削減困難なコードクローンの差異に関する調査は，それらのメトリクスを比較することで行った．クローンメトリクスを分析する目標は，その得られた結果を基に削減可能なコードクローンと削減困難なコードクローンの差異を詳らかにする足がかりとする立ち位置である．例えば，クローンセットに属するコードクローンの数を表すクローンメトリクス *POP* で得られた知見がコードクローンのリファクタリング可能性に影響するのか調査をした場合，最終的にその影響が確認できた場合，*POP* を用いることで簡易的なコードクローンのリファクタリング可能性の判定に有用である可能性がある．

分析における脅威

クローンメトリクスの分析調査はソフトウェア保守に基づいたコードクローンの研究に有用であると考えられる．ただし，コードクローンのリファクタリング可能性はソースコードを記述するプログラミング言語の様々な制約を受ける．例えば，ある特定のプログラミング言語におけるクローンメトリクスがもたらすリファクタリング可能性への影響を分析で得たとする．しかし，その言語がもつ独特の特性により得られた傾向である可能性がある．すなわち，調査を行ったプログラミング言語以外の言語では，その傾向とは異なる可能性がある．また，同一言語であっても，プログラミング言語は調査した段階における文法に，開発者の利便性を考慮した新たな文法が追加されたり，潜在的な脆弱性を補うために既存の文法が変更されたりする可能性がある．そのため，ここでの分析調査は最終的な結果には至らない．

分析対象

図 1 で示した 9 つのクローンメトリクスを分析の対象であるクローンメトリクスである．すなわち，*LEN*，*POP*，*NIF*，*RAD*，*RNR*，*TKS*，*LOOP*，*COND*，*McCabe* である．またこれらのクローンメトリクスの計測に用いたツールは CCFinderX である．CCFinderX では，検出したコードクローンに対して，これら 9 つのクローンメトリクスを測定する．その測定結果は gemX と呼ばれる CCFinderX の GUI を用いることで確認できる．

ウェルチの t 検定

本研究では，2 つの母集団から差異を調査する統計学的手法の 1 つとして，ウェルチの t 検定を用いた．t 検定とは，実証したい仮説とは反対の意味を持つ帰無仮説を正しいと仮定し，統計量が t 分布とよばれる連続した確率分布に従うことを利用した統計学の手法の 1 つである．一般的に，スチューデントの t 検定と呼ばれる手法は，2 つの独立した母集団の分散が等しいと仮定できる場合に用いられる．2 つの独立した母集団の分散が等しくないと仮定される場合にはウェルチの t 検定を用いる．

本研究では，削減可能なクローンセットと削減困難なクローンセットから得られる母集団は独立しており，また分散が等しくないとみられるので，ウェルチの t 検定を用いる．ここでいう独立していない母集団とは次のようなものである．例えば，1 つの計測対象に対して，ある事象の前後における計測結果をそれぞれ標本にした場合には，2 つの標本間には対応関係があるとみなせる．削減可能なクローンセットと削減困難なクローンセットにはそのような対応関係がないため独立した関係であるといえる．

ウェルチの t 検定の手法は次のように行う．

1. 帰無仮説，および対立仮説を立てる．
2. 統計量として t 値をもとめ，p 値へ変換する．
3. p 値と有意水準を比較して，下回っていれば，帰無仮説を棄却する．反対に，上回っていれば，対立仮説を棄却する．

1. では，立証ししたい仮説を対立仮説として立てる．その反対の意味を持つ仮説を帰無仮説と呼ぶ．帰無仮説の必要性は，t 検定における帰無仮説の棄却は可能ではあるが，帰無仮説の正当性を述べることができないことにある．そのため対立仮説だけではなく、帰無仮説も同時に立てる必要がある．

2. では t 値を求める．それぞれの母集団の平均，分散，サンプルサイズを $\bar{X}_i$，s_i，N_i と

する．このとき，t は次のように定義されている．

$$t = \frac{\bar{X_1} - \bar{X_2}}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

これは，2 つの集団が有意な差がある条件を数式にしている．すなわち，1 つの母集団の平均に差があるほど，また分散が小さいほど，そしてサンプルサイズが大きいほど有意な差があると考えられるからである．

t 値を求めると，次にこれを有意確率と呼ばれる p 値に変換する．変換には確率分布を扱う必要があるが，一般的には計算を行うツールなどを用いる．ここでは，その解法自体は省略する．

3．では，t 検定から得られる解釈を意味している．有意水準とは，仮説が成り立つ確率 p が有意であるための基準のことである．有意確率がこの有意水準を下回る時，帰無仮説を棄却して，2 つの母集団間に有意差が認められることになる．反対に，有意確率が有意水準を上回る時は，対立仮説を棄却するだけにとどまる．

分析手法

次の手順をもって，クローンメトリクスを分析した．

- Step1 CCFinderX で測定されたクローンセットごとのクローンメトリクスに対して，そのクローンセット ID を追跡することで，削減可能なクローンセットと削減困難なクローンセットのクローンメトリクスを求める．
- Step2 分類された削減可能なクローンセットと削減困難なクローンセットをそれぞれ母集団として，ウェルチの t 検定を行うため，帰無仮説と対立仮説を立てる．
- Step3 有意水準 $\alpha = 0.05$ として，2 つの母集団から得られる p 値を比較して，有意差を調べる．

Step1 では，7 つの OSS から検出されたクローンセットを削減可能なクローンセットと削減困難なクローンセットに分類された後，各クローンセットのクローンメトリクスの値をひもづけるためにクローンセット ID で追跡する．各プロジェクトから検出されたクローンセットはそのプロジェクト内では固有の ID を持つ．そのため，クローンセットをプロジェクトごとで管理することでクローンセット ID を通じた識別が可能である．

Step2 では，削減可能なクローンセットと削減困難なクローンセットをそれぞれ母集団とする．ここでの作業は，プロジェクトごとに管理していたクローンセットをまとめることにある．ここからの工程において，固有のクローンセットやコードクローンを追跡する必要が

ないので，このように 1 つの集団にまとめてしまっても問題ない．また，本研究では，削減可能なクローンセットと削減困難なクローンセットから得られる母集団は独立しており，また分散が等しくないとみられるので，ウェルチの t 検定を用いることとした．

我々は削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスの母集団には，有意な差があると仮定した．すなわち，帰無仮説と対立仮説は次のようにした．

- 帰無仮説

  削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスには差がない．

- 対立仮説

  削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスには差がある．

Step3 では，有意確率と有意水準を比較する．有意確率の算出には Microsoft Excel 2016 を用いた．有意水準は一般的に用いられる数値として $\alpha = 0.05$ を採用した．

分析結果

表 9 は，削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスのウェルチの t 検定の結果を示している．各クローンメトリクスごとに有意確率を算出している．この有意確率が有意水準 $\alpha = 0.05$ より小さければ，2 つの母集団間に有意差があることが認められる．9 つのクローンメトリクスについて，8 つについて有意差が認められた．認められていない 1 つのメトリクスは NIF であった．

ウェルチの t 検定では，両側検定と呼ばれる 2 つの母集団間の差にあらかじめ方向性を設けない検定が行われる．方向性とは，一方の母集団の平均がもう一方の母集団の平均より大きい（または小さい）ことを仮説に設けることである．また，そのような検定を片側検定と呼ぶ．片側検定は，一方の標本がもう一方の標本より必ず小さくなることはない（あるいは大きくなることはない）という前提のもと，仮説が立てられる場合が多い．

本研究では，削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスに対して，そのような方向性を持っているのかさえ不透明であったため，片側検定ではなく両側検定を実施した．しかし，どちらの方向によっているのかある程度の指標を方向として表 9 に載せている．この指標は表 10 に基づいて決定している．

表 10 は，削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスの各平均値を示している．有意差が認められたクローンメトリクスの

表 9: クローンメトリクスの有意確率における両側検定による有意差の有無とその方向

クローンメトリクス	有意確率	有意差の有無	方向
LEN	$6.1 * 10^{-32}$	有	小
POP	0.035	有	小
NIF	0.134	無	-
RAD	$4.9 * 10^{-16}$	有	小
RNR	$5.0 * 10^{-104}$	有	大
TKS	$1.5 * 10^{-5}$	有	大
LOOP	$3.3 * 10^{-5}$	有	大
COND	$2.9 * 10^{-19}$	有	小
McCabe	$8.6 * 10^{-33}$	有	小

うち，この平均が削減可能なクローンセットのクローンメトリクスの平均値のほうが大きい場合は，表 9 の方向は大と示した．反対に削減可能なクローンセットのクローンメトリクスのほうが小さい場合は，小と示している．

有意差の認められた 8 つのクローンメトリクスのうち，*LEN*，*POP*，*RAD*，*COND*，*McCabe* の 5 つのクローンメトリクスにおいて，削減可能なクローンセットのほうが削減困難なクローンセットより平均値が小さく，*RNR*，*TKS*，*LOOP* の 3 つは平均値が大きかった．

考察

削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスの有意差は 8 つのクローンメトリクスで認められた．*LEN* や *RNR* は実際の数値でみても，削減可能なクローンセットと削減困難なクローンセットの間で大きな差があったが，t 検定で有意差が認められたことでその傾向が強いと考えられる．これらのメトリクスの有意差から推定できる削減可能と評価されやすいコードクローンの特徴は次のようなものである．

- *LEN* について，コード片のトークン数が小さいほど削減可能である．
- *POP* について，1 つのクローンセットに属するコードクローンの個数が少ないほど削減可能である．
- *RAD* について，コードクローン間の共通の親ディレクトリまでの距離が小さいほど削減可能である．

表 10: 削減可能なクローンセットと削減困難なクローンセットのそれぞれから得られたクローンメトリクスの各平均値

クローンメトリクス	削減可能	削減困難
LEN	87.16	120.83
POP	2.88	3.03
NIF	1.74	1.76
RAD	0.55	0.60
RNR	0.78	0.60
TKS	17.17	16.42
LOOP	0.20	0.13
COND	1.22	1.75
McCabe	1.43	1.89

- *RNR* について，コード片の繰り返しが少ないほど削減可能である．
- *TKS* について，コード片に含まれるトークンの種類が多いほど削減可能である．
- *LOOP* について，コード片に含まれる反復処理が多いほど削減可能である．
- *COND* について，コード片に含まれる条件分岐が少ないほど削減可能である．
- *McCabe* について，反復処理の数と条件分岐の数の合計は少ないほど削減可能である．

LEN は先に述べたように，トークン数はソースコード行数に比例しているものとみなしている．そのため，削減可能なクローンセットはソースコード行数が小さなものとみなすことができる．*POP* に関しては，少なからず貪欲法の影響を受けている可能性がある．貪欲法ではクローンセットあたりの削減可能ソースコード量が大きな順にリファクタリングの候補にしている．そのため，オーバーラップしているクローンセットの中では *POP* が小さいほうが有利になる可能性がある． 2 つのメトリクス *LOOP* と *COND* は大小関係の方向性が異なる．*COND* は大きいほど，JDeodorant の前提条件の 7 に違反する可能性が高まる可能性があるためであると考えている．*LOOP* が数が多い時が有利であるのは，反復処理単位のコード片のリファクタリングは削減可能になりやすいためであると考えている．

ただし，*NIF* に有意差が見られなかったが，t 検定の非対称的な性質により，コードクローンがもつファイルの個数が削減可能なクローンセットと削減困難なクローンセットに有意差な差をもたらさないとはいえないことに留意したい．

4.4 複数のメソッドを持つコードクローンの分割

動機

本研究では，JDeodorant をクローンペアのリファクタリング可能性を評価するために用いている．JDeodorant の機能はクローンペアのリファクタリング可能性をバイナリ的な評価を出力する．すなわち，リファクタリング可能であるかリファクタリング困難であるかという評価方法である．しかし，JDeodorant のリファクタリング評価機能は，このバイナリ的評価以外の判定を下す場合がある．すなわち，コードクローン自体に何らかのエラーがあった場合である．その中の 1 つにコード片に複数のメソッドが含まれていた場合にエラーとなることが判明した．CCFinderX では，複数のメソッドが含まれているようなコードクローンを検出する．しかし，そのようなコードクローンは JDeodorant において，エラーとなり評価そのものをしない．そこで，我々は追加的な実験として，JDeodorant のリファクタリング可能ではないクローンペアから，複数のメソッドを持たないように自動処理を行った場合の削減可能ソースコード量への影響を調査した．

手順

複数のメソッドが存在するコードクローンを分割して，1 つのコードクローンに最大で 1 つのメソッドしか持てないようにするのが目標である．

次の手順で行う．

- Step1 複数のメソッドを持つ可能性があるコードクローンを発見する．
- Step2 各コードクローンに対して，コード片中に存在するメソッドを識別して分割する．
- Step3 分割したコードクローンを JDeodorant の分析 1 段階目で得られた xls ファイルと同じフォーマットになるように出力する．
- Step4 分割したコードクローンの情報を JDeodorant の 2 段階目の分析にかけて，リファクタリング可能性を評価し，提案手法を用いて削減可能ソースコード量を求める．

Step1 複数のメソッドを持つ可能性があるコードクローンの発見

Step1 では，まず，複数のメソッドを持つ可能性があるコードクローンを発見する必要がある．提案手法では，発見すべきコードクローンは JDeodorant の 2 段階目の分析結果の際に，リファクタリング可能ではないコードクローンとして除外されている．JDeodorant の 2 段階目の分析結果ファイルの中で，提案手法では，クローンペアのリファクタリング可能性の評価結果を取得するために tree.tsv ファイルを用いた．tree.tsv ファイルで取得できる

リファクタリング可能性の評価結果は true または false で表現されているが，この評価対象にはエラーが原因で評価されなかったクローンペアは含まれていない．

そこで，もう 1 つの JDeodorant の出力結果ファイルである-analyzed.xls ファイルを用いる．図 4 にあるように，Details の列にはセルが緑色または赤色が指定されている．そのようなセルには，色と同時にどのクローンペアの組み合わせに対するリファクタリング可能性を評価しているのか出力されている．その評価順番は位置によってきまっている．クローンペアはクローンセットごとにまとまって出力されていて，上三角の形に位置が決まる．もっとも左上に割り当てられるのはそのクローンセットに属するコードクローンのうち，1 番目と 2 番目のクローンペアとなる．

エラーが発生した際には，このあるべき文字列情報が欠落する．この情報を欠落したセルを持つコードクローンがエラーが出たために JDeodorant にリファクタリング可能性を評価されなかったと判断できる．複数のメソッドを持つコードクローンはこの欠落したコードクローンに含まれることになる．ここから複数のメソッドを持つコードクローンを探索せず，次の工程に進む．

Step2 メソッド単位の分割

図 6 は複数のメソッドを持つコードクローンを分割する様子を表している．クローンセット a はメソッド 1 とメソッド 2 を含むコードクローンとメソッド 3 とメソッド 4 を含むコードクローンを持つ．これをメソッド単位で分割するとは，1 つのコードクローンに 2 つ以上のメソッドが含まれないようにコードクローンを分割するということである．すなわち，メソッド 1 とメソッド 3 が対応関係にあり，メソッド 2 とメソッド 4 が対応関係にあるため，それぞれをコードクローンとするような 2 つのクローンセット b，c に割り当てなおすことである．

まずは，コードクローンを分割するにあたって必要な情報について確認する．コードクローンが存在するソースコードやオフセットの所在情報は-analyzed.xls ファイルでわかる．ただし，メソッド名とメソッドシグネチャは空白である．これらの情報は分割時にソースコードを読み込むことで取得できる．メソッドの判別は，次の 2 段階を経る．

1. コードクローンが存在するソースコードに対して，構文的な解釈と中括弧 ({ と }) の対応関係を合わせることで，ファイル構造を解析する．
2. 解析したファイル構造とコードクローンの所在情報をもとに，各コードクローンを分割する．

1. におけるファイル構造の解析とは，ここでは，どの行数がクラスの外，フィールド変数，メソッド内なのかを簡単に判断していることを意味している．例えば，Java ソースコー

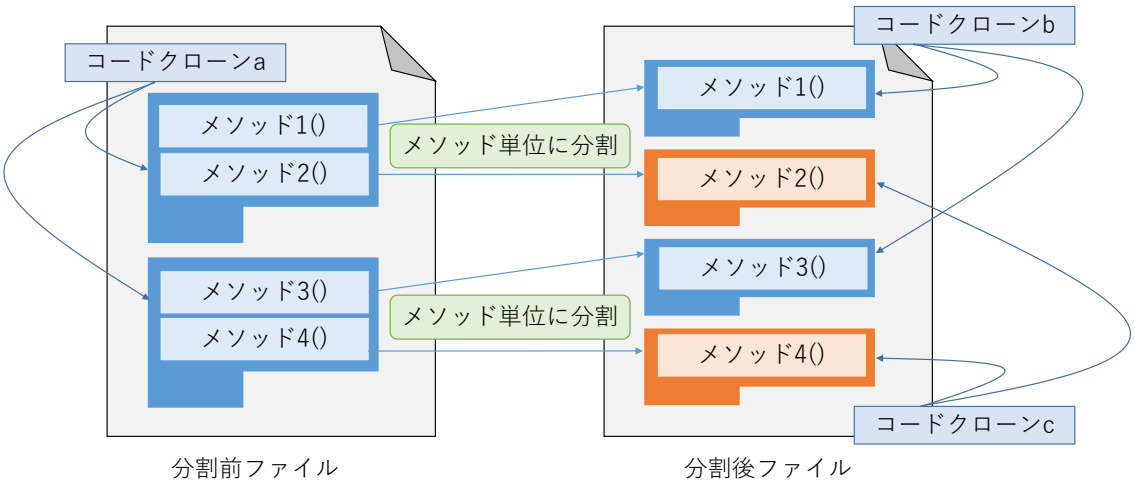

図 6: 複数のメソッドを持つコードクローンの分割

ドはメソッドの宣言に省略可能なアクセス修飾子の後に，メソッドの型，メソッド名などのシグネチャ情報を特定の順番で記述する必要がある．この文法を利用して，ソースコードの構造を解析する．ここで，メソッド内にある行に関しては，そのメソッドの名前も記憶しておいた．

2. では，行数単位で判別したファイル構造と，開始行番号と終了行番号とそれぞれのオフセットが判明しているコードクローンの所在情報から，コードクローンの各行がどのメソッドに属しているのかが判明する．これによって，コードクローンをメソッド単位に分割することが可能になる．なお，単一のメソッドにしか属していないコードクローンも同様の処理をされる．この場合，分割後も単一もメソッドに属することになる．

Step3 分割後のコードクローンの所在情報の出力

分割したコードクローンは JDeodorant で 2 段階目の分析ができるように 1 段階目の分析結果として出力される xls ファイルのフォーマットに合わせる必要がある．必要な情報の中でとりわけ作業が必要な項目は，クローンセット ID，メソッド名，メソッドシグネチャ，開始行番号と終了行番号，開始オフセットと終了オフセットである．また，クローン間の位置関係についての記述も適時書きかえる必要がある．

クローンセット ID は，入力したクローンセットの中から最大のクローンセット ID に対し

て，分割後のクローンセットにはその最大のクローンセット ID 以上の数値を用いれば良い．

メソッド名やメソッドシグネチャ，行番号，オフセットの情報は，ファイル構造を把握する段階で判明しているので，それを利用すればよい．ただし，メソッドシグネチャだけは特殊な記号が用いられている．例えば，void 型の引数なしのメソッドのシグネチャは，()V で表現される．() 内には，引数が入る．V は void 型を表している．他にも int 型は I，boolean 型は B，またそれ以外は String ならば QString; のように Q と; で挟んで表現する．

Step4 分割したコードクローンに対する削減可能ソースコード量の算出

分割したコードクローンに対する削減可能ソースコード量は提案手法にしたがって算出する．ただし，分割したコードクローンは本来検出したコードクローンとは異なる所在やトークン長を示すために，開発者の意図とは異なる結果を生み出す可能性が存在する．そのため，本研究では，複数のメソッドを持つコードクローンの分割は補助的な立ち位置にとどめている．

表 11 は複数のメソッドを持つコードクローンを分割した際の削減可能ソースコード量の変化を示している．削減可能ソースコード量は表 8 のものと同じである．この数値を基準に，追加される行数を観察する．対象行数は，複数のメソッドを持つ可能性のあるコードクローンの行数である．また，トータルの削減可能ソースコード量が，元の削減可能ソースコード量にリファクタリング可能な分割したコードクローン行数を加えたものである．また，表 11 における () 内は全コードクローン行数に対する割合である．{} 内は元の削減可能ソースコード量との差分を表している．

Apache Ant や JFreeChart は影響が大きく，1,000 行以上の追加が見込まれている．それに対して，これら以外のプロジェクトではあまり効果が見られなかった．特に，JMeter では差分は見られなかった．

削減可能ソースコード量の差分にあまり変化が見られなかったのは，JDeodorant の挙動によるところが大きい．すなわち，複数のメソッドを持つコードクローン以外にもエラーが発生する条件が存在すると考えている．その原因の解明については今後の課題である．

4.5 手動のリファクタリングとの比較

動機

JDeodorant 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は，JDeodorant に依存する見積りである．しかし，この数値の信頼度は，JDeodorant の正当性が不可欠となる．そこで，この章では，JEdit でリファクタリング可能と評価されたクローンセット 14 個を対象に，手動でリファクタリングを実行して，その削減量を計測した．そして，その手

表 11: コード片分割による削減可能ソースコード量の増加

プロジェクト名	コード片分割前の削減可能ソースコード量	対象行数	コード片分割後の削減可能ソースコード量
Apache Ant	3,429 (5.7)	44,728 (67.1)	4,046 {+1,017} (6.7)
Columba	584 (10.7)	766 (14.1)	635 {+51} (11.6)
JMeter	385 (6.1)	311 1(49.0)	385 {+0} (6.1)
JEdit	136 (6.6)	351 (17.0)	143 {+7} (6.9)
JFreeChart	9,700 (5.0)	23,332 (11.9)	11,450 {+1,750} (5.9)
JRuby	2,161 (3.2)	580,334 (87.1)	2,206 {+45} (3.2)
Apache Xerces	5,533 (5.8)	87,707 (92.5)	5,564 {+31} (5.8)

動による計測値と自動的に見積もった削減可能ソースコード量を比較して，削減可能ソースコード量の正当性を確認する．

手法

JDeodorant でリファクタリング可能と評価された JEdit に含まれる 14 個のクローンセットを，図 2 で説明した手法に基づいて手動でリファクタリングを行った．ここで，計測する削減行数は，削減可能ソースコード量に基づいて同様の計算方法を用いて測定した．すなわち，リファクタリング前のクローンセット全体の行数からリファクタリング後の共通メソッドの行数と置換された呼び出し文の和を引いたものとなる．

結果

表 12 は JEdit で削減可能と評価されたクローンセット 14 個を対象に，手動でリファクタリングしたコードクローンの削減量と自動的に算出した削減可能ソースコード量を示している．自動的に推定された削減可能ソースコード量は 136 行に対して，手動による削減行は 131 行であった．すなわち，5 行ほどの差が存在することが分かった．この誤差については，次の節において議論する．

誤差が発生した原因に関する考察

5 行の誤差が発生した原因としては，次の 2 つが考えられる．

- 返り値のために return 文が必要な場合がある．

表 12: 14 個のクローンセットを対象とした手動によるコードクローンの削減量と自動的に推定した削減可能ソースコード量の比較

手動による削減量	推定された削減可能ソースコード量
131	136

- 共通メソッドや呼び出し文の開始行や終了行を開発者のコーディングスタイルに一致させた場合に，複数行にわたる場合がある．

1 つ目は共通メソッドに返り値が必要となり，その返り値を返すための return 文を追加するために誤差が生まれてしまう場合になる．14 個のクローンセットのうち，1 個のクローンセットに見られた．この誤差を解決するには，共通メソッドとして抽出してきたコード片を分析して，必要な返り値とその型を取得する必要があると考えられる．

2 つ目は開発者のコーディングスタイルを考慮した結果，共通メソッドの開始行数が複数行にわたる場合が見られたために誤差が生まれた．本調査の場合は，一部の開発者がソースコード中の中括弧のみを改行して 2 行に渡るコーディングをしていたため，誤差が生まれた．この場合以外にも，extends 文や implement 文，長すぎる引数などを改行するコーディングスタイルは考えられる．これらを解決するためには，開発者ごとのコーディングスタイルを考慮する必要がある．あるいは，解決を図るのではなく，改行そのものはプログラムの動作の本質そのものには影響がないと考えて，コーディングスタイルを考慮しない削減可能ソースコード量を提供することも考えられる．

図 7 は誤差が発生したコードクローンの一部とそのリファクタリングするにあたって作成した共通メソッドである．ソースコードの一部は冗長であるために中略している．元のコード片は，Object 型の変数 source を getSource() メソッドで取得した後，そこに代入されている変数に応じて処理が分岐している．また，このコードクローン箇所の後の処理で，ふたたび変数 source を利用するために source の変数を返す必要がある．そのため，共通メソッドには，その変数 source を返すために return 文を追加している．また，このソースコードを作成した開発者はメソッドの開始行数を中括弧で改行するコーディングスタイルである．そのため，開発者のコーディングスタイルを尊重して，作成した共通メソッドの開始行でも中括弧を改行させている．

4.6 考察

JDeodorant が評価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は，検出されたコードクローン全体のおよそ 5%から 6%であることが判明した．また，リファク

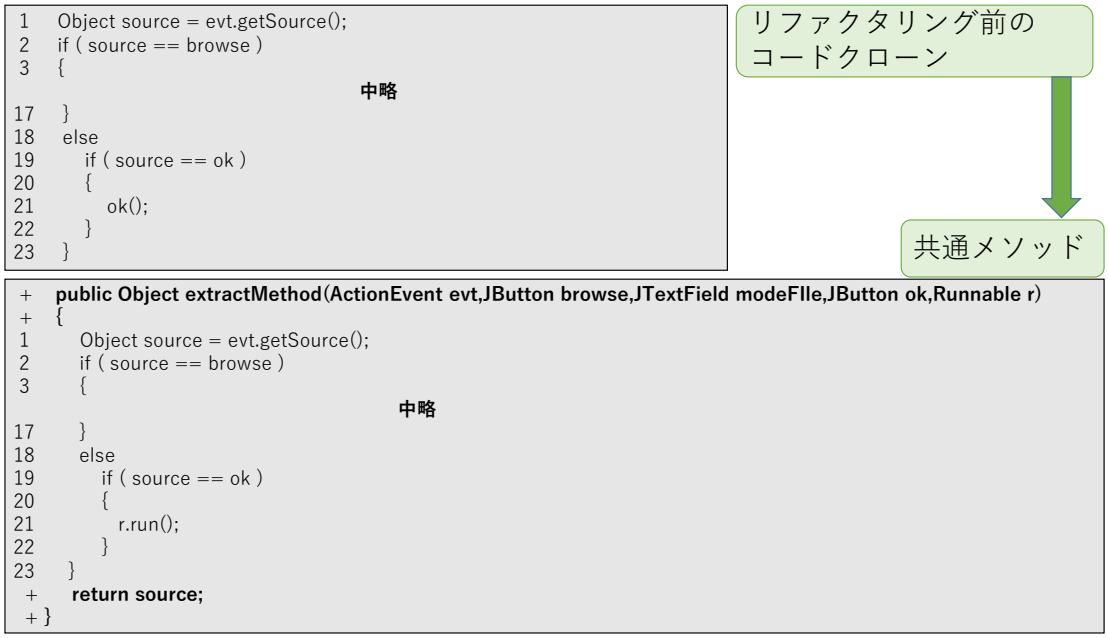

図 7: 誤差が発生したコードクローンのリファクタリング例

タリング可能なコードクローンの特徴は，9 つのクローンメトリクスを調査することで，大まかではあるがある傾向を突き止めることができた．そして，複数のメソッドを持つコード片分割をした後による削減可能ソースコード量が部分的にではあるが，増加することが確認された．また，手動によるリファクタリング結果は 131 行となり，推定された削減可能ソースコード量 136 行と比較しても遜色ない結果となった．発生した誤差については，それぞれ適切な対処を検討する必要がある．そして，この傾向は JEdit だけではなく他のオープンソースソフトウェアについても同様に得られると考えている．

JDeodorant は開発者にとって安全で簡単なコードクローンのリファクタリングを提供している．そのため，4.2 章の結果や 4.3 で得られた傾向は，変更する恐れがある．しかしながら，検出したコードクローンを開発者がほとんど加工をせずに得られる削減可能ソースコード量としては重要な知見を得られたと考える．

5　まとめ

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 7 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．

また，削減可能なコードクローンと削減困難なコードクローンからそれぞれ測定したクローンメトリクスを分析した結果，9 つのクローンメトリクスのうち 8 つのクローンメトリクスについて有意差が認められた．

削減困難なコードクローンの中には，コードクローン検出ツールの特性によって複数のメソッドを含んでいるコードクローンが JDeodorant のリファクタリング可能性を判定する前提条件に違反する場合が確認された．しかし，必ずしもすべてのプロジェクトで削減可能性を変更するわけではない．オープンソースソフトウェアによっては，コード片分割の影響を受けないものが存在していることも確認した．

削減可能ソースコード量の正当性を評価するために，手動のリファクタリングを行った．その結果，削減可能ソースコード量の比較的強い正当性と算出手法の改善すべき点が判明した．改善点については今後の課題としたい．

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．1 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

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, 御多忙の中，常に適切な御指導及び御助言を賜りました. また，研究室の環境が研究に大変専念させていただきやすかったこと，深く感謝を申し上げます.

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 准教授には進捗報告を通じて，研究について多くの御助言を頂きました. また，研究だけではなく学業に関しましても多くの御助言をいただきましたことを，心より感謝します.

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 准教授には本研究について多くの御指導をしていただきました．また，本論文を執筆にあたり数々の御助言を賜りましたことを，深く感謝しています.

最後に，様々な御指導，御助言等を頂き, 研究生活を支えてくださった, 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] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, pp. 104–113, 2012.

[2] Salah Bouktif, Antoniol Giuliano, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. *Proceedings of the 8th annual conference on Genetic and evolutionary computation Proceedings of GECCO 2006*, pp. 1885–1892, 2006.

[3] CCFinderX. `http://www.ccfinder.net`.

[4] Buford Edwards III, Yuhao Wu, Makoto Matsushita, and Katsuro Inoue. 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.

[5] Mark Harman, McMinn Phil, Jerffeson, Teixeira de Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. *Empirical Software Engineering and Verification*, pp. 1–59, 2012.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. *Proceedings of the 29th international conference on Software Engineering Procceedings of ICSE 2007*, pp. 96–105, 2007.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 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.

[12] 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.

[13] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: memory comparison-based clone detector. *Proceedings of the 33rd International Conference on Software Engineering Proceedings of ICSE 2011*, pp. 301–310, 2011.

[14] M.Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[15] Mark O'Keeffe and Mel O Cinneide. 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.

[16] Simian. `http://www.harukizaemon.com/simian`.

[17] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, Vol. 41, No. 11, pp. 1055–1090, 2015.

[18] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Assessing the refactorability of software clones. *International Conference on Software Engineering (ICSE'2017)*, pp. 60–70, 2017.

[19] 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.

[20] 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.

[21] 肥後芳樹, 吉田則裕, 楠本真二, 井上克郎. 産学連携に基づいたコードクローン可視化手法の改良と実装. 情報処理学会論文誌, Vol. 48, No. 2, pp. 811–822, 2007.