

## コードクローンを対象としたリファクタリング支援環境

肥後 芳樹<sup>†</sup> 神谷 年洋<sup>††</sup> 楠本 真二<sup>†</sup> 井上 克郎<sup>†</sup>

Refactoring Support Environment Based on Code Clone Analysis

Yoshiki HIGO<sup>†</sup>, Toshihiro KAMIYA<sup>††</sup>, Shinji KUSUMOTO<sup>†</sup>, and Katsuro INOUE<sup>†</sup>

あらまし 作業の効率を悪化させている一要因として、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同一、または類似したコード片のことである。例えば、あるコード片にバグが含まれていた場合、そのコード片のコードクローンすべてについて修正の是非を考慮する必要がある。このような理由によりソフトウェアからコードクローンを取り除くことはソフトウェアの保守性や複雑度などの面からみて有効である。これまでに、いくつかのコードクローン集約手法が提案されているが、解析時間コストが高いなどの理由により、大規模なソフトウェアに対しては適用が難しかった。本論文では、大規模ソフトウェアに対しても適用可能なコードクローンの集約支援手法の提案を行う。具体的には、実用的な時間でソースコード中から集約に適したコードクローンを検出し、メトリックスを用いてコードクローンの特徴を定量化し、それに基づきコードクローンの絞込みを行う。本手法を用いることによって、各コードクローンに適した集約方法を提示することで、ユーザは効率的なリファクタリング作業を行うことができる。また提案手法をリファクタリング支援環境 Aries として実装し、適用実験を行うことで、本手法の有用性を確認した。

キーワード リファクタリング, コードクローン, ソフトウェア保守

## 1. ま え が き

近年、コードクローンがソフトウェア保守を困難にしている一つの要因といわれている。コードクローンとはソースコード中に存在する同一、または類似したコード片のことである。コードクローンが生成される原因としては様々な理由が考えられるが、その最も大きな原因の一つとしてコピーアンドペーストによる修正、拡張作業が挙げられる。あるコード片にバグが含まれていた場合、そのコード片のコードクローンすべてに対して修正の是非を考慮する必要がある。このような作業は、特に大規模ソフトウェアでは非常に手間のかかる作業である。したがってコードクローン検出の効率化はソフトウェア開発・保守工程の改善において有効である。これまでにコードクローンを自動的に検出するための様々な手法が提案されている [2], [4], [5], [10], [13], [15]。

その手法の一つとして、我々はコードクローン検出ツール CCFinder [10] と分析環境 Gemini [16] を開発してきている。ユーザは Gemini を用いることによりコードクローンの解析、ソースコードの修正を容易に行うことができる。様々なコードクローンの把握・管理手法が提案されている一方で、ソフトウェアからコードクローンを取り除く研究はそれほど活発に行われてはいない。これまでにいくつかの提案がされているが [11], [12], それらは時間的なコストが非常に高いものであったりと、大規模ソフトウェア開発・保守現場での適用は困難である。

本論文では、実用的な時間でソースコード中からリファクタリングに適したコードクローンを検出し、更に、検出したコードクローンの特徴をメトリックスを用いて定量化する手法を提案する。そして、提案手法に基づき、リファクタリング支援環境 Aries の試作を行う。最後に適用実験を行い、Aries の有効性を評価する。

## 2. 準 備

## 2.1 コードクローンの定義

あるトークン列中に存在する二つの部分トークン列

<sup>†</sup> 大阪大学大学院情報科学研究科, 豊中市  
Graduate School of Information and Science Technology,  
Osaka University, Toyonaka-shi, 560-8531 Japan

<sup>††</sup> 科学技術振興機構さきかけ  
PRESTO, Japan Science and Technology Agency, Japan

$\alpha, \beta$  が等価であるとき,  $\alpha$  と  $\beta$  は互いにクローンであるという. またペア ( $\alpha, \beta$ ) をクローンペアと呼ぶ.  $\alpha, \beta$  それぞれを真に包含するいかなるトークン列も等価でないとき,  $\alpha, \beta$  を極大クローンと呼ぶ. また, クローンの同値類をクローンセットと呼ぶ. ソースコード中でのクローンを特にコードクローンという [9].

### 2.2 CCFinder

CCFinder [10] はプログラムのソースコード中に存在する極大クローンを検出し, その位置をクローンペアのリストとして出力する. 検出されるコードクローンの最小トークン数はユーザが前もって設定できる.

CCFinder のコードクローン検出手順 (ソースコードを読み込んで, クローンペア情報を出力する) は以下の四つの STEP からなる.

STEP1 ( 字句解析 ): ソースファイルを字句解析することによりトークン列に変換する. 入力ファイルが複数の場合には, 個々のファイルから得られたトークン列を連結し, 単一のトークン列を生成する.

STEP2 ( 変換処理 ): 実用上意味をもたないコードクローンを取り除くこと, 及び, 些細な表現上の違いを吸収することを目的とした変換ルールによりトークン列を変換する. 例えば, この変換により変数名は同一のトークンに置換されるので, 変数名が付け替えられたコード片もコードクローンであると判定することができる.

STEP3 ( 検出処理 ): トークン列の中から指定された長さ以上一致している部分をクローンペアとしてすべて検出する.

STEP4 ( 出力整形処理 ): 検出されたクローンペアについて, ソースコード上での位置情報を出力する.

### 2.3 CCSHaper

CCShaper [7], [8] は CCFinder の検出したコードクローンから, 構造的なまとまりをもった部分をリファクタリングに適したコードクローンとして抽出する. 図 1 はその例を示している. 図 1 では, A と B の二つのコード片が示されている. A と B それぞれの灰色の部分は, その部分が A と B の間の最大長のコードクローンであることを示している. コード片 A ではいくつかのデータがリスト構造の先頭から順に連続して格納されている. 一方コード片 B では, リスト構造の後方から順に連続してデータが格納されている. これら二つのコード片には, リスト構造を扱う共通のロジック ( for 文 ) が含まれているが, コード片

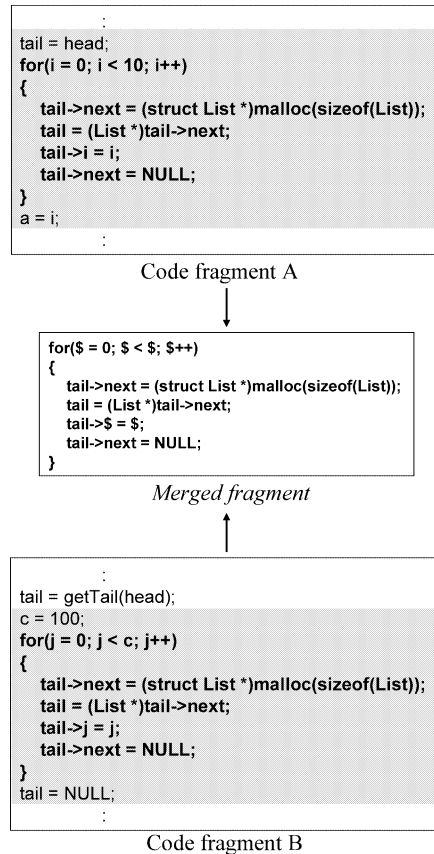


図 1 コードクローン集約の例  
Fig. 1 Example of merging two code fragments.

の最初と最後には, 偶然クローンとなった部分 ( 代入文 ) も含まれてしまっている. 集約を目的とした場合, 灰色の部分全体よりも for 文のみをコードクローンとして抽出の方が望ましい. CCSHaper ではこのような場合, 灰色で示されたコードクローンから構造的なまとまりをもった部分, つまり for 文の部分のみを抽出する.

### 3. 提案手法

上述のように, CCFinder が検出するコードクローンには, リファクタリングの適用対象にできないものも多く含まれる. CCSHaper ではこのようなコードクローンの大部分を取り除いてはいるが, ユーザが自ら抽出されたコードクローンの除去方法を考えなければならないため, リファクタリングを行うには非効率的である. 本論文ではこの除去方法の決定支援を行うために, メトリックスを用いたコードクローンの絞込み

手法を提案する．以下ではまず，3.1において，本手法で提案するメトリックスを説明し，3.2では，提案したメトリックスを用いてのコードクローンの絞込み方法を，例を用いて説明する．なお，本手法では，2.3で述べたような言語における構造的なまとまりをもったコードクローンを対象としている．

### 3.1 集約支援を目的としたメトリックス

これまでに様々なリファクタリングパターン [6] が提案されており，以下の7種類のリファクタリングパターンがコードクローンを除去するために用いることができる．

- Extract Class,
- Extract Method,
- Extract SuperClass,
- Form Template Method,
- Move Method,
- Parameterize Method,
- Pull Up Method.

本手法ではこれらのパターンを用いてコードクローンの集約を行うリファクタリングの支援を目的とする．これらのパターンは，コードの一部を新たなモジュールとして抽出するものと，モジュールの移動を行うものに分類できる．

まず，新たなモジュールとして抽出を行う場合を“Extract Method”を例にとって考える．本来は“Extract Method”は長過ぎるメソッドや，複雑な処理の一部分に対して適用することによって，コードの可読性，保守性を向上させることができる．しかし，コードクローンに対して適用することにより，重複したコード片を集約することも可能である．コード片を新たなメソッドとして再定義することになるため，抽出部分は周囲との結合度が低いことが望ましい．つまり，抽出部分の外側で定義された変数を抽出部分でできるだけ用いていないことが望ましい．もしそのような変数を用いていた場合は，抽出したメソッドの引数として与える，あるいはメソッドの戻り値として返す必要がある．抽出部分とその周囲の結合度を計測するために  $NRV(S)$  (the Number of Referred Variables) と  $NSV(S)$  (the Number of Substituted Variables) の二つのメトリックスを定義した．ここでは，クローンセット  $S$  は  $n$  個のコード片  $f_1, f_2, \dots, f_n$  を含んでおり，コード片  $f_i$  では  $s_i$  個の外部定義の変数を参照しており， $t_i$  個の外部定義の変数に対して代入を行っているとする．このとき  $NRV(S)$  と  $NSV(S)$  はそ

れぞれ次の式で表される．

$$NRV(S) = \frac{1}{n} \sum_{i=1}^n s_i, \quad NSV(S) = \frac{1}{n} \sum_{i=1}^n t_i,$$

直観的には， $NRV(S)$  はクローンセット  $S$  に含まれる各コード片内で参照されている外部定義変数の平均数を示し，同様に  $NSV(S)$  は代入が行われている変数の平均数を示す．

次に，モジュールの移動を行う場合を“Pull Up Method”を例にとって考える．“Pull Up Method”とは，ある親子クラス関係が存在した場合に，子クラスに存在するメソッドを親クラスに引き上げることである．もし共通の親クラスをもつ複数の子クラスに重複したメソッドが存在した場合は，それらを共通の親クラスに引き上げることによって集約を行うことが可能である．つまり重複したメソッドを含むクラスは共通の親クラスを継承している必要がある．そのため，クローンセットのクラス階層内における位置関係を計測する．これについては，メトリックス  $DCH(S)$  (the Dispersion of Class Hierarchy) を定義する．既に示したように，クローンセット  $S$  はコード片  $f_1, f_2, \dots, f_n$  を含んでいるとする．またクラス  $C_i$  はコード片  $f_i$  を含んでいるクラスとする．もしクラス  $C_1, C_2, \dots, C_n$  が共通の親クラスをもつ場合は，その共通の親クラスの中で，最もクラス階層的に下位に位置するクラスを  $C_p$  で表すとする．また  $D(C_k, C_h)$  はクラス  $C_k$  と  $C_h$  のクラス階層における距離を表すとする．このとき，

$$DCH(S) = \max \{D(C_1, C_p), \dots, D(C_n, C_p)\}$$

と表される．直観的には，メトリックス  $DCH(S)$  はクローンセット  $S$  に含まれる各コード片間のクラス階層内における最大の距離を示す．例えば，クローンセット  $S$  中のすべてのコード片が一つのクラス内に存在する場合は  $DCH(S)$  の値は 0，あるクラスとその直接の子クラス内に存在する場合は  $DCH(S)$  の値は 1 となる．例外的に，コードクローンが存在するクラスが共通の親クラスをもたない場合は  $DCH(S)$  の値は -1 とする．このメトリックスは，JDK のクラスライブラリ等の修正不可能なクラスを除外したクラスを対象として計算される．これにより，分析対象のソフトウェア内に存在するメソッドを修正不可能なクラスに引き上げようとする場合は， $DCH(S)$  の値は -1 となり，そのようなリファクタリングが不可能で

あることが分かる。

### 3.2 メトリックスを用いた絞込み

本節では、提案したメトリックスを用いてのコードクローンの絞込み方法を例を用いて説明する。絞込みでは前節で提案した三つのメトリックスに加え、 $LEN(S)$ ,  $POP(S)$ ,  $DFL(S)$  [16] の三つのメトリックスも用いる。各メトリックスの簡単な説明を以下に示す。

$LEN(S)$ : クローンセット  $S$  に含まれるコード片のトークン数の平均値を表す。この値が大きいクローンセットは除去の候補となる。

$POP(S)$ : クローンセット  $S$  に含まれるコード片の数を表す。この値が大きいほど同形のコード片がより多くソースコード中に存在することになり、除去の候補となる。

$DFL(S)$ : クローンセット  $S$  を再構築した場合に減少するトークン数の予測値を表す。ここでの再構築とは、 $S$  内のコード片集合から抜き出した共通のロジックを実装するサブルーチンを作り、各コード片をそのサブルーチンの呼出しに置き換えることである。 $DFL(S)$  は、再構築前のトークン数 (すなわち、 $S$  に含まれるすべてのコード片のトークン数の和) から、再構築後のトークン数 (すなわち、すべてのサブルーチン呼出しのトークン数の和+共通ロジックを実装するサブルーチンのトークン数) を引いたものとして定義される。 $DFL(S)$  は  $S$  を除去した場合のサイズ面における効果の指針としてとらえることができるため、この値の大きいクローンセットは除去の候補となる。

以降、これらのメトリックスを用いた絞込みについて、例を用いて説明する。

#### (1) “Pull Up Method”

例えば、以下のような条件が考えられる。

- (PC1) 対象となる単位はメソッド本体、
- (PC2)  $DCH(S)$  の値が 1 以上。

“Pull Up Method” はメソッドが対象であるので、条件 (PC1) が必要である。また、重複したメソッドを含むクラスが共通の親クラスを継承している必要があることから条件 (PC2) が必要である。この条件でクローンセットの絞込みを行った場合、抽出されたクローンセットは以下の四つのグループに分類される。

(PG1) 単純に親クラスに引き上げるのみで集約可能なクローンセット。

(PG2) 外部定義の変数を引数として追加した後、親クラスに引き上げることによって集約可能なクローン

セット。

(PG3) 外部定義の変数を引数として追加し、更に返り値として返す処理を追加した後、親クラスに引き上げることによって集約可能なクローンセット。

(PG4) その他、すなわち、(PG1)~(PG3) 以上の工夫が必要であるクローンセット。

#### (2) “Extract Method”

“Extract Method” を行う際の条件としては例えば、以下のものが挙げられる。

- (EC1) 対象となる単位は文単位、
- (EC2)  $DCH(S)$  の値が 0、
- (EC3)  $NSV(S)$  の値が 1 以下。

“Extract Method” とはメソッド内のコード片に対して適用されるので、(EC1) が必要である。また、すべてのコードクローンが同一のクラス内に存在する場合は容易に集約が可能であるので、条件 (EC2) を考慮している。コードクローンの内部において、外部定義変数に対して代入を行っている場合は、その変数を引数として与え、返り値として返し、メソッドの呼出し元に反映させなければならない。このような変数が複数あった場合は新たなデータクラスを定義し、そのオブジェクトを介して値を受け渡す必要がある。もしこのような変数が一つの場合は単に return 文を用いて返すだけでよく、容易に集約を行うことができるので、条件 (EC3) を考慮している。

この条件でクローンセットの絞込みを行った場合、抽出されたクローンセットは以下の四つのグループに分類される。

(EG1) 単純にコードクローンをメソッドとして括り出すのみで集約可能なクローンセット。

(EG2) コードクローン内部で使用されている外部定義の変数を抽出するメソッドの引数とすることによって集約可能なクローンセット。

(EG3) コードクローン内部で使用されている外部定義の変数を抽出するメソッドの引数とし、更に返り値として返すことによって集約可能なクローンセット。

(EG4) その他、すなわち、(EG1)~(EG3) 以上の工夫が必要なクローンセット。

#### (3) その他

例として上記の二つの条件を述べたが、絞込みに使える条件はこれらの条件に限らず様々なものが考えられる。例えば “Extract Method” を行う場合は、結合度を  $NSV(S)$  だけでなく  $NRV(S)$  も併用して用いることができる。これによりある個数以下の引数しか

もたないようなメソッドとして括り出すことができる。また他のパターンに対しては、例えば、“Move Method”を行う場合はクラス階層を考慮する必要がないので、メトリックス  $DCH(S)$  では絞込みを行わず、メソッドとクラスとの結合度を計る  $NRV(S)$  や  $NSV(S)$  で絞込みを行うことが考えられる。また、除去することによって10,000 トークン以上の減少が見込まれるクローンセットのみを  $DFL(S)$  によって絞り込む、手間をかけずにリファクタリングを行いたいので  $POP(S)$  が5以下のクローンセットのみを対象とするなどといったことが可能である。つまり本手法では、用いる六つのメトリックスについて、ユーザが興味のあるメトリックスの上限や下限、またはその両方を用いて絞込みを行うことができる。

#### 4. リファクタリング支援環境：Aries

##### 4.1 概要

提案手法を、リファクタリング支援環境 Aries として実装した。現在のところ対象は Java 言語としている。また、構造的なコードクローンの検出には、既存のコードクローン検出ツール CCFinder と CCShaper を用いている。つまり、CCFinder を用いて、トークンの列としてのコードクローンを検出し、その中に含まれる構造的なまとまりを、CCShaper を用いて抽出している。Java 言語を対象としているため抽出する構造的なまとまりは以下の12種類である。

宣言 : class { }, interface { }  
 メソッド : メソッド本体, コンストラクタ,  
           スタティックイニシャライザ  
 文 : if, for, while, do, switch,  
       try, synchronized

図2(a), 2(b)はAriesのスナップショットである。ユーザは図2(a)のMain Windowにおいてリファクタリング対象となるクローンセットの絞込みを行うことができる。また個々のクローンセットについてより詳細な情報をClone Set Viewerを用いて得ることができる。

##### 4.2 インタフェース

Ariesの各インタフェースを簡単に紹介する。

###### 4.2.1 Metric Graph View

図3を例にとってMetric Graph Viewを説明する。Metric Graph Viewではメトリックスごとに1本の並行座標軸が用意される。また、クローンセットごと

に1本の折れ線が描画される。この例では二つのクローンセット  $S_1$  と  $S_2$  が描画されている。図3(a)はMetric Graph Viewの初期状態を表している。各並行座標軸には初期値として、その並行座標軸が表すメトリックス値をすべて含むように上限と下限が設定される。クローンセットはそのメトリックス値のすべてが各並行座標軸の上限と下限の間に収まっている場合は選択状態となり、それ以外の場合は非選択状態となる。図中の網がかかった部分が各並行座標軸の上限と下限の間を表している。つまり、初期状態では、すべてのクローンセットの各メトリックス値はその並行座標軸の上限と下限の間に収まっており、選択状態となっている。ユーザは各並行座標軸の上限、下限付近でマウスの左ボタンを押し、そのまま上下にドラッグすることによって、各上限、下限を変更可能である。例えば、図3(b)ではユーザがメトリックス  $DCH(S)$  の上限を下げた状態を表している。この状態では  $DCH(S_2)$  が上限より大きくなってしまっており、クローンセット  $S_2$  は非選択状態となっている。Metric Graph Viewを用いたクローンセットの選択・非選択状態はClone Set Listに反映される。

###### 4.2.2 Checkbox of Used Variable

Checkbox of Used Variableでは、ユーザはどの種類の変数をメトリックス  $NRV(S)$ ,  $NSV(S)$  の要素としてカウントするのかを決定することができる。変数は以下の6種類に分類されている。

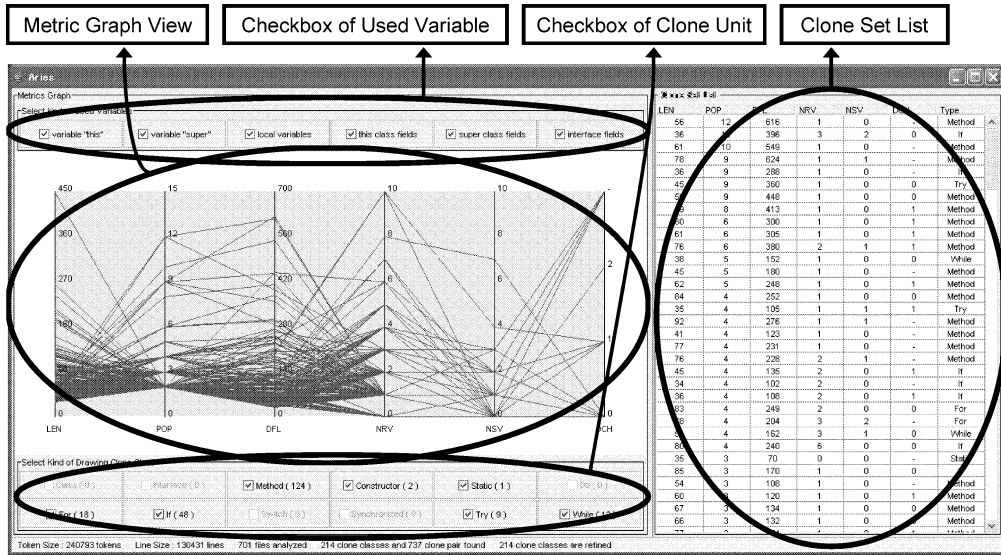
- 同クラスのフィールド変数,
- 親クラスのフィールド変数,
- インタフェースのフィールド変数,
- this 参照,
- super 参照,
- ローカル変数.

###### 4.2.3 Checkbox of Clone Unit

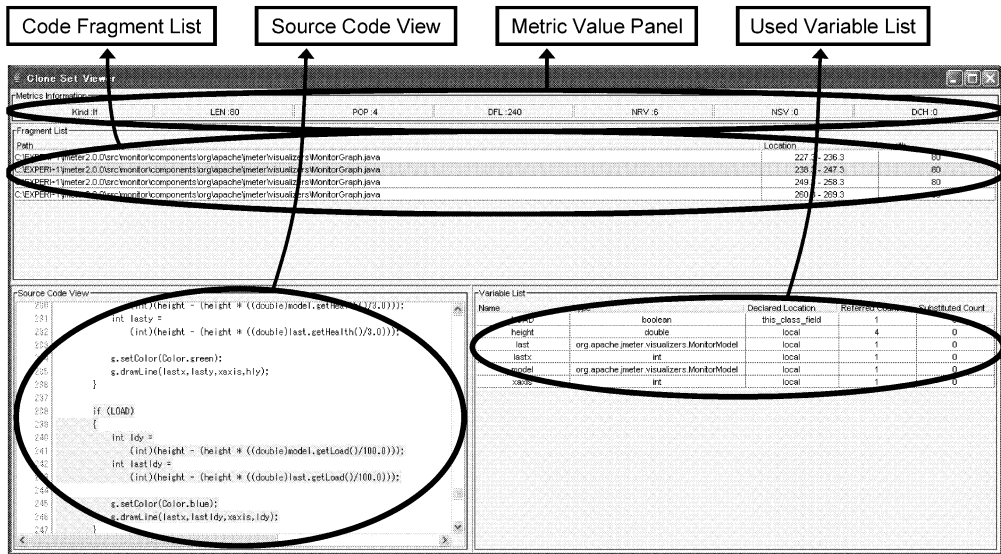
Checkbox of Clone Unitでは、ユーザはどの単位のクローンセットをMetric Graph Viewにおける選択の対象とするのかを決定することができる。選択できる単位は4.1で述べた12種類である。

###### 4.2.4 Clone Set List

Clone Set ListはMetric Graph Viewにおいて選択状態にあるクローンセットの一覧を表示する。Metric Graph Viewにおいて非選択状態となっているクローンセットはこのリストには表示されない。つまり、ユーザがMetric Graph Viewにおいて興味のあるメトリックスを用いて絞込みを行った結果、すべてのメ



(a) Main window



(b) Clone set viewer

図 2 Ariès のスナップショット  
Fig. 2 Snapshots of Ariès.

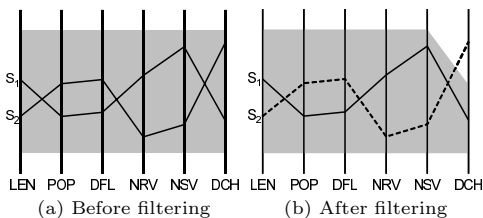


図 3 Metric Graph View を用いた絞り込み

Fig. 3 Refining clone sets using Metric Graph View.

トリックス値が絞り込みの範囲内となっているクローンセットのみがこのリストに表示される。例えば, Metric Graph View が図 3(a) の状態の場合, Clone Set List にはクローンセット  $S_1$  と  $S_2$  が表示されているが, 図 3(b) の場合はクローンセット  $S_1$  のみが表示されることになる。このように Metric Graph View での選択・非選択状態を用いることによってユーザは, 絞り込みに該当するクローンセットの一覧を Clone Set

List 上で得ることができる。また、このリストに表示されているクローンセットは各メトリクス値に基づいて昇順・降順にソートが可能である。そしてこのリストに表示されている任意のクローンセット上でマウスの左ボタンをダブルクリックすることにより、そのクローンセットに関するより詳細な情報を示す Clone Set Viewer (図 2(b)) が表示される。

#### 4.2.5 Metrics Value Panel

Metrics Value Panel はそのクローンセットの各メトリクス値を示す。

#### 4.2.6 Code Fragment List

Code Fragment List はそのクローンセットに含まれるコード片の一覧を示す。各コード片について、そのコード片を含むファイルへのパス、ファイル内での位置(開始行, 開始列, 終了行, 終了列), そのコード片のトークン数が表示される。

#### 4.2.7 Source Code View

Source Code View は Code Fragment List において選択されたコードクローン片周辺のソースコードを示す。コードクローンは強調して表示される。

#### 4.2.8 Used Variable List

Used Variable List は Code Fragment List において選択されたコード片内で用いられているが、その外部で定義されている変数の参照, 代入回数を示す。

## 5. 適用実験

### 5.1 概要

Aries の有効性を評価するために、オープンソースの Java ソフトウェアである Ant [1] に対して適用実験を行った。実験では、まず Ant のソースコードから構造的なコードクローンを検出し、次に、前述の絞込み条件に従って、二つのリファクタリングパターン “Pull Up Method” と “Extract Method” を適用するのに適したコードクローンを選出する。選出されたコードクローンについて、リファクタリングパターンが適用できるかを評価する。

実験の対象となった Ant のソースコードは 627 ファイルからなり、総行数は約 18 万行である。この適用実験では、検出する構造的なコードクローンの最小トークン数は 30 とした。Ant から構造的なコードクローンを検出するのに要した時間は約 1 分であった<sup>(注1)</sup>。この実験では Ant に対して、“Pull Up Method” と “Extract Method” の適用を試みた。コードクローン検出の結果 Aries は Ant から 154 個のクローンセッ

トを検出した。

また、“Pull Up Method” と “Extract Method” の適用を試みたクローンセットの数は以下のとおりである。

All detected clones	154
“Pull Up Method”	20
“Extract Method”	59

“Pull Up Method” と “Extract Method” を適用するための絞込み条件は 3.2 で説明したものと同様である。以下 5.2, 5.3 では、絞り込んだ結果についてより詳細に述べる。

### 5.2 “Pull Up Method”

本節では “Pull Up Method” の条件の適用結果について述べる。既に述べたように、絞り込んだ結果、20 個のクローンセットを抽出した。この 20 個のクローンセットのに含まれるすべてのコードクローンのソースコードを閲覧し、3.2 で述べた (PG1)~(PG4) の四つのグループに分類した。

まず、(PG1) に分類されたクローンセットは全く存在しなかった。

10 個のクローンセットが (PG2) に分類された。図 4 はその一例を示している。このコードクローンでは、“this” が省略されている、なぜなら “getCommentFile” は同クラス内に定義されたメソッドだからである。このメソッド内では変数 “this” と “FLAG\_COMMENTFILE” が外部定義である。つまり、このコードクローンに対して “Pull Up Method” を適用する場合はこれら二つの変数を引数として追加する必要がある。

2 個のクローンセットが (PG3) に分類された。図 5

```
private void getCommentFileCommand(Commandline cmd) {
    if (getCommentFile() != null) {
        /* Had to make two separate commands here because
           if a space is inserted between the flag and the
           value, it is treated as a Windows filename with
           a space and it is enclosed in double quotes (").
           This breaks clearcase.
        */
        cmd.createArgument().setValue(FLAG_COMMENTFILE);
        cmd.createArgument().setValue(getCommentFile());
    }
}
```

図 4 (PG2) に分類されたコードクローンの例  
Fig.4 Example of pull up method in (PG2).

(注1): Aries の実行環境: OS FreeBSD4.9, CPU Xeon 2.8 GHz, メモリ 4 GByte.

はその一例を示している。このコードクローンにおいて代入を行われている変数 “map” は外部で定義されている (“setError” は共通の親クラスで定義されたメソッド)。よって、このメソッドを親クラスに引き上げるには、この変数を引数として追加し、更に代入結果をメソッド呼出し元に反映させるために return 文を追加する必要がある。

8 個のクローンセットが (PG4) に分類された。図 6 はその一例を示している。このコードクローンでは、メソッド “checkOptions” を呼び出している。メソッド “checkOptions” は同クラス内で定義さ

れている (“getProject”, “getViewPath”, “getLocation” は共通の親クラスで定義されている)。また、このメソッド呼出しの引数となっている変数 “commandLine” はこのコードクローンの内部で定義されており、このクローンに対して “Pull Up Method” を適応することは困難である。しかし、“checkOptions” は各子クラスで定義されていることから、“Form Template Method” [6] を適応することが可能であると考えられる。手順としては、このコードクローンを親クラスにそのままの形で引き上げ、次に親クラスに “checkOptions” の抽象メソッドを定義する。

### 5.3 “Extract Method”

絞込みの結果、59 個のクローンセットを抽出した。抽出したすべてのコードクローンのソースコードを閲覧し、3.2 で述べた (EG1)~(EG4) の四つのグループに分類した。

3 個のクローンセットが (EG1) に分類された。図 7 はその一例を示している。このコードクローンでは外部定義の (ローカル) 変数を全く用いていなかった。このコードクローンはそのままメソッドとして抽出するのみでリファクタリング可能である。

34 個のクローンセットが (EG2) に分類された。図 8 はその一例を示している。このコードクローンでは外部定義の変数 “javacopts” と “genieTask” を参照している。“javacopts” は同クラス内に定義されたフィールドであるのに対し、“genieTask” はローカル変数であるため、このコードクローンをメソッドとして抽出するためには、この変数を引数とする必要がある。

15 個のクローンセットが (EG3) に分類された。図 9 はその一例を示している。このコードクローンでは外

```
public void verifySettings() {
    if (targetdir == null) {
        setError("The targetdir attribute is required.");
    }
    if (mapperElement == null) {
        map = new IdentityMapper();
    } else {
        map = mapperElement.getImplementation();
    }
    if (map == null) {
        setError("Could not set <mapper> element.");
    }
}
```

図 5 (PG3) に分類されたコードクローンの例  
Fig. 5 Example of pull up method in (PG3).

```
public void execute() throws BuildException {
    CommandLine commandLine = new CommandLine();
    Project aProj = getProject();
    int result = 0;

    // Default the viewpath to basedir if it is not specified
    if (getViewPath() == null) {
        setViewPath(aProj.getBaseDir().getPath());
    }

    // build the command line from what we got. the format is
    // cleartool checkin [options...] [viewpath ...]
    // as specified in the CLEARTOOL.EXE help
    commandLine.setExecutable(getClearToolCommand());
    commandLine.createArgument().setValue(COMMAND_CHECKIN);

    checkOptions(commandLine);

    result = run(commandLine);
    if (Execute.isFailure(result)) {
        String msg = "Failed executing: " +
            commandLine.toString();
        throw new BuildException(msg, getLocation());
    }
}
```

図 6 (PG4) に分類されたコードクローンの例  
Fig. 6 Example of pull up method in (PG4).

```
if (!isChecked()) {
    // make sure we don't have a circular reference here
    Stack stk = new Stack();
    stk.push(this);
    dieOnCircularReference(stk, getProject());
}
```

図 7 (EG1) に分類されたコードクローンの例  
Fig. 7 Example of extract method in (EG1).

```
if (javacopts != null && !javacopts.equals("")) {
    genieTask.createArg().setValue("-javacopts");
    genieTask.createArg().setLine(javacopts);
}
```

図 8 (EG2) に分類されたコードクローンの例  
Fig. 8 Example of extract method in (EG2).

```

if (iSaveMenuItem == null) {
    try {
        iSaveMenuItem = new MenuItem();
        iSaveMenuItem.setLabel("Save BuildInfo To Repository");
    } catch (Throwable iExc) {
        handleException(iExc);
    }
}

```

図 9 (EG3) に分類されたコードクローンの例  
Fig. 9 Example of extract method in (EG3).

```

if (name == null) {
    if (other.name != null) {
        return false;
    }
} else if (!name.equals(other.name)) {
    return false;
}

```

図 10 (EG4) に分類されたコードクローンの例  
Fig. 10 Example of extract method in (EG4).

部定義の変数 “iSaveMenuItem” に対して代入処理を行っている。よって、このコードクローンをメソッドとして抽出するには、この変数を抽出するメソッドの引数とし、更に返り値として返す必要がある。

7個のクローンセットが (EG4) に分類された。図 10 はその一例を示している。このコードクローンでは return 文が使用されている。これにより、このコードクローンをメソッドとして抽出する場合は、上記以上の工夫が必要である。

## 6. 関連研究

Komondoor ら [11] や Krinke ら [12] はプログラム依存グラフを用いる手法を提案している。この手法では、グラフ上での類似部分がコードクローンとして検出される。この手法ではソースコード中の制御依存やデータ依存を解析しているため、非常に精度が高いのが特徴である。更に、reordered クローンや intertwined クローンなどの特殊なコードクローンを検出することができる。このようなコードクローンは本手法では検出することが不可能である。ただし、グラフ構築の時間コストが  $O(n^2)$  である<sup>(注2)</sup>ため、大規模ソフトウェアに対しての適用は難しい。それに対して本手法では、簡単な意味解析までにとどめているため、たとえ対象が大規模であっても実用的な時間で解析を行うことが可能である。実際に、J2SDK Standard Edition のソースコード (約 127 万行) に対して Aries

を実行したところ約 4 分 20 秒で構造的なクローンを検出することができた<sup>(注3)</sup>。

また Balazinska ら [3] は、サイクロマチック数 [14] など 21 種類のメトリックスを用い、各メトリックス値の一致や近似に基づきコードクローン検出を行っている。各メトリックスは関数・メソッドに対して計測されるので、検出されるコードクローンもその単位に限定される [13]。本手法では、関数・メソッド単位だけではなく対象言語の構造的なまとまりすべてをコードクローンの単位としているので、Balazinska らの手法よりもリファクタリング候補のコードクローンをより多く検出すると期待できる。

## 7. む す び

本論文では、コードクローンを対象としたリファクタリング手法を提案した。また、提案手法をリファクタリング支援ツール Aries として実装した。Aries ではプログラム依存グラフを構築するなどの複雑な解析を行っておらず、簡単な意味解析までにとどめているため、大規模なソフトウェアに対しても実用的な時間で適用可能である。また、適用実験では、Aries を用いて、オープンソースのソフトウェアである Ant のソースコードから、リファクタリング手法の “Pull Up Method” と “Extract Method” を適用可能なコードクローンを特定することができた。

現在の解析はリファクタリングの可能性について述べているが、積極的にすべきかの判断はしていない。今後は、ソフトウェアの品質の面からリファクタリングの是非を判断するように拡張を行う予定である。また、影響波及解析を行い、リファクタリングによる影響波及個所の特定を試みる。これにより、より効率的なリファクタリング支援環境を提案することができると思われる。

謝辞 本研究は一部、文部科学省リーディングプロジェクト「eSociety 基盤ソフトウェアの総合開発」、文部科学省科学研究費若手研究 (B) (課題番号: 15700031) の支援を受けている。

## 文 献

- [1] Ant, <http://ant.apache.org>, 2003.
- [2] B.S. Baker, “Parameterized duplication in strings:

(注2):  $n$  はソースコード中の文や式の数。

(注3): Aries の実行環境: OS FreeBSD4.9, CPU Xeon 2.8 GHz, メモリ 4 GByte。

- Algorithms and an application to software maintenance,” SIAM J. Comput., vol.26, no.5 pp.1343–1362, 1997.
- [3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, “Advanced clone-analysis to support object-oriented system refactoring,” Proc. the 7th Working Conference on Reverse Engineering, pp.98–107, Brisbane, Australia, Nov. 2000.
- [4] 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, Bethesda, Maryland, March 1998.
- [5] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” Proc. International Conference on Software Maintenance 99, pp.109–118, Oxford, England, Aug. 1999.
- [6] M. Fowler, Refactoring: Improving the design of existing code, Addison Wesley, 1999.
- [7] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “On software maintenance process improvement based on code clone analysis,” Proc. 4th International Conference on Product Focused Software Process Improvement, pp.185–197, Rovaniemi, Finland, Dec. 2002.
- [8] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎, “コードクローン解析に基づくリファクタリングの試み,” 情処学論, vol.45, no.5, pp.1357–1366, May 2004.
- [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, Sept. 2001.
- [10] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multi-linguistic token-based code clone detection system for large scale source code,” IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, July 2002.
- [11] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” Proc. the 8th International Symposium on Static Analysis, pp.40–56, Paris, France, July 2001.
- [12] J. Krinke, “Identifying similar code with program dependence graphs,” Proc. the 8th Working Conference on Reverse Engineering, pp.301–309, Stuttgart, Germany, Oct. 2001.
- [13] J. Mayland, C. Leblanc, and E.M. 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, Monterey, California, Nov. 1996.
- [14] T.J. McCabe, C.W. Butler, “Design complexity measurement and testing,” Commun. ACM, vol.32, pp.1415–1425, Dec. 1989.
- [15] M. Rieger and S. Ducasse, “Visual detection of duplicated code,” Proc. ECOOP Workshop on Experiences in Object-Oriented Re-Engineering, pp.75–76, Brussels, Belgium, July 1998.

- [16] 植田泰士, 神谷年洋, 楠本真二, 井上克郎, “開発保守支援を目指したコードクローン分析環境,” 信学論 (D-I), vol.J86-D-I, no.12, pp.863–871, Dec. 2003.  
(平成 16 年 5 月 21 日受付, 8 月 10 日再受付)



肥後 芳樹

平 14 阪大・基礎工・情報中退。平 16 同大大学院博士前期課程了, 現在, 同大学院博士後期課程 1 年。コードクローン分析の研究に従事。情報処理学会会員。



神谷 年洋 (正員)

平 8 阪大・基礎工・情報中退。平 13 同大大学院博士課程了。現在, 科学技術振興機構さきがけ研究者。博士(工学)。オブジェクト指向関連技術, ソフトウェア保守(メトリックス, コードクローン), 認知科学に関する研究に従事。情報処理学会, 電気学会, IEEE 各会員。



楠本 真二 (正員)

昭 63 阪大・基礎工・情報卒。平 3 同大大学院博士課程中退。同年同大・基礎工・情報・助手。平 8 同大講師。平 11 同大助教授。平 14 阪大・情報・コンピュータサイエンス・助教授。博士(工学)。ソフトウェアの生産性や品質の定量的評価, プロジェクト管理に関する研究に従事。情報処理学会, IEEE, JFPUG, PM 各会員。



井上 克郎 (正員)

昭 54 阪大・基礎工・情報卒。昭 59 同大大学院博士課程了。同年同大・基礎工・情報・助手。昭 59–61 ハワイ大マノア校・情報工学科・助教授。平元阪大・基礎工・情報・講師。平 3 同学科・助教授。平 7 同学科・教授。博士(工学)。平 14 阪大・情報・コンピュータサイエンス・教授。ソフトウェア工学の研究に従事。情報処理学会, 日本ソフトウェア科学会, IEEE, ACM 各会員。