Vol.2011-SE-173 No.8
2011/7/21

リファクタリング支援のための コードクローン間の識別子名の対応関係分析

工藤 良介[†1]　伊達 浩典[†1]
石尾 隆[†1]　井上 克郎[†1]

ソフトウェアの保守コストを高める要因の 1 つとして，コードクローンの存在が挙げられる．コードクローンを解消するためのリファクタリング手法があるが，この手法がどの程度のコードクローンに対して適用できるかは判明していない．本論文では，クローンセット間の識別子名の違いに注目することで，リファクタリングの適用可能範囲を調査する．

Analysis of Correspondence of Identifier between Code Clones to Support Refactoring

Ryosuke Kudo,[†1] Hironori Date,[†1] Takashi Ishio[†1]
and Katsuro Inoue [†1]

It is said that the existence of code clones is a factor of increasing maintenance cost. As one of the ways to remove code clones, refactoring is a well-known approach. However it is unclear how many clones this way can be applied to. This research aims to make it clear by focusing the difference of identifies among clone sets.

1. まえがき

ソフトウェアの保守コストを増大させる要因のひとつとして，コードクローンの存在が挙げられる．コードクローンとはソースコード中に存在する同一，または類似のコード片を持つコード片のことであり，主にコピーアンドペーストによるコード記述によって生成される．互いにクローンとなっているコード片の集合（クローンセット）の中のひとつのコード片にバグが存在した場合，そのクローンセットに含まれる他のコード片すべてについて不具合の有無を判定し，修正しなくてはならない[7)]．このようなソフトウェアの確認作業は自動化が困難であり，開発者の目視による作業となるため，開発者の大きな負担となっている．そのため，可能ならばコードクローンを除去することが望まれる．

コードクローンを解消するためのリファクタリング手法のひとつとして，クローンとなっているコード片を 1 つのメソッドとして独立させ，クローンであった箇所をその呼び出しに置換する方法がある．しかし，クローンセット中のコード片は同一であるとは限らず，コード片間の違いを吸収するような工夫を求められる．このとき，各コード片が持つ差異の種類によって，集約の難易度は大きく異なる．たとえば，クローンとなっているコード片がそれぞれ異なる変数を参照していた場合は，それらの変数を引数として新しく作成したメソッドに渡せばよいので，集約は容易に可能であると考えられる．一方で，各コード片が異なるメソッドを呼び出している場合は，動的束縛を用いてメソッドの呼び出しを切り替えるなどの工夫が必要であり，より複雑な手順を必要とする．

本研究では，クローンごとの識別子名の違いという観点からクローンを分類し，集約の期待できるクローンセットを抽出しリファクタリング手法の適用支援を行うことを目的とする．具体的には，識別子を変数，メソッドなどに分類し，どの種類の識別子名が変更されたかという観点からクローンセットをラベル付けし，それぞれの特徴や傾向について調査した．

オープンソースソフトウェア 2,142 個を対象として，変数名，メソッド名，型名などの識別子について，コードクローンとなっているコード片の間で，どれだけ差異があるかを調査した．その結果，全コードクローン中の約 4 分の 1 は，完全一致あるいは変数名の差異のみであり，集約の期待できるクローンであることが判明した．また，変数名などの識別子は，95 %が完全一致あるいはコード片の間で 1 対 1 の対応関係を取ることが明らかになった．よって，リファクタリング手法において，識別子の対応関係を前提とした機械的な編集の適用が期待できる．

以下 2 章では本研究の背景について述べる．3 章では調査の内容について説明する．4 章では識別子の分類やその差異の数を抽出する際に用いた手法について述べる．5 章では得られた結果とその考察について述べる．最後に 6 章では本研究のまとめと今後の課題について述べる．

†1 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

2. 背　　景

コードクローンと一言で言っても，コードクローンを検出するツールにより様々な定義がある．本研究が対象とするコードクローンを明確にするために，本章ではまず，コードクローンとその分類を説明し，次に，コードクローンを取り除くために行われるリファクタリングの手法を解説する．

2.1 コードクローン

コードクローンとは，あるプログラムにおいて互いに類似したコード片の組あるいはコード片の集合のことである．コードクローンは，多くの場合，開発者がソースコードを意図的に複製して再利用することによって生じる[6]．しかし，コードクローンとなっているコード片同士は，互いに完全一致するとは限らず，何らかの差異を含むことがある．Bellon らは，この差異の種類に着目してコードクローンを以下の 3 つのタイプに分類している[2]．

タイプ 1: 空白や改行などのコーディングスタイルの違いを除き，完全に一致するコードクローン

タイプ 2: 変数名，型名，メソッド名などの識別子名や，型名などを表す一部の予約語のみが異なるコードクローン

タイプ 3: タイプ 2 に加えて，一部に文の挿入や削除が行われたコードクローン

あるプログラムが，コードクローンを多数含んでいたとしても，それ自体がプログラムの実行に悪影響をもたらすわけではない．しかし，プログラム中のコードクローンの増加は，ソースコードの保守性，変更容易性に悪影響を与える．例えば，複製したコードにバグが含まれていることが明らかになった場合，複製されコードクローンとなったコード片すべてに適切な修正を施す必要があり，変更容易性が損なわれ保守性が低下する．そのため，特に長期的に保守する計画のあるソフトウェアにおいて，コードクローンを可能な限り取り除きたいという要求がある．

様々なコードクローン検出ツールのうち，実用的なものとしては，トークン単位でコードクローンを検出する CCFinder がある．CCFinder によって検出されるクローンは，ソースコード中に登場する識別子名やリテラルを「パラメータ化」することでその違いを吸収したクローン（Parameterized Clone）であり，タイプ 1 及びタイプ 2 のクローンがこれに該当する．

開発者は，単にコードクローン検出ツールを用いてコードクローンの位置を把握するだけでなく，出力されたコードクローンについて詳細に分析を行い，もし必要があればクローンを取り除くための作業を実施することになる．

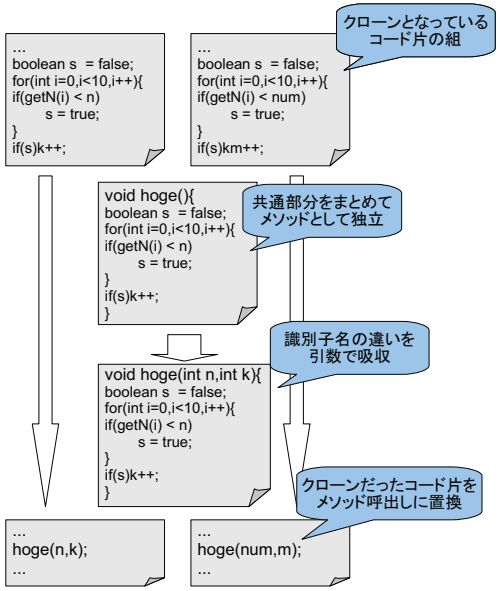

図 1 コードクローンを解消するリファクタリング例
Fig. 1 Example of refactoring for code clones

2.2 リファクタリング

リファクタリングとは「プログラムの外部から見た動作を変えずにソースコードの内部構造を整理すること」である[3]．リファクタリングは，プログラム自体の動作を変えるものではないが，コードの見通しをよくして将来的な修正の要求に耐えられるようにすることを主な目的としている．

2.2.1 コードクローンのリファクタリング

コードクローンの存在はソフトウェアの保守を困難にするため，コードクローンを解消するための手法が研究されている[5]．

ここでは，互いにクローンとなっているコード片をメソッドとして抽出することにより，コードクローンを解消するリファクタリングを紹介する．**図 1** のように，コードクローンを解消するためのリファクタリングは，次の 3 つのステップで行われる．

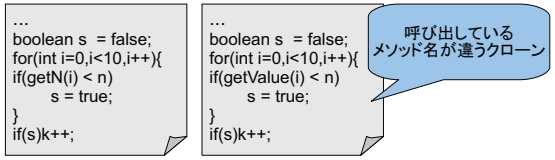

図 2 メソッド名が異なる場合
Fig. 2 Case that methods' names are different

(1) 共通部分を新たな一つのメソッドとして独立させる.
(2) 引数を設定するなどしてクローン間の識別子名の違いを吸収する.
(3) クローンとなっているコード片を，メソッド呼び出しに置換する.

以下，このようなステップを踏んで行うリファクタリングを「コードクローンの集約」と呼ぶ.

2.2.2 集約することが困難なコードクローン

2.1 節で述べたように，コードクローンは 3 つのタイプに分類できる．タイプ 1 に分類されるコードクローンは互いに完全に一致しているが，タイプ 2 やタイプ 3 に分類されるコードクローン間には，何らかの差異が存在していることになる．差異を含むコードクローンの集約を行うためには，差異の部分の修正，除去を検討する必要があるため[1)]，完全一致のコードクローンの集約と比較すると差異を含むコードクローンの集約は困難となる.

以下にコードクローンの集約が困難な場合の例を示す.

クローン間で呼び出しているメソッド名が異なる場合

図 2 で示すように，クローン間で呼び出しているメソッドの識別子が異なる場合，コードクローンの集約は困難である.

クローン間で型名が異なる場合

クローンとなっているコード間で使用されているデータ型が異なる場合，通常，コードクローンの集約は困難であるが，総称型（Generic Type）の適用や，継承関係によるクラス名の統一によって集約が可能な場合もある．たとえば**図 3** のコードでは，変数の型名は親クラスに統一可能である.

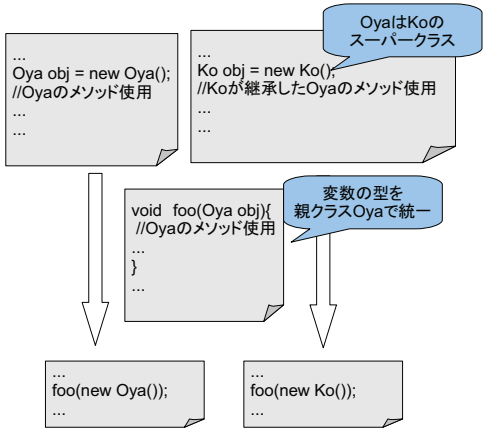

図 3 型名が異なる場合
Fig. 3 Case that types' names are different

3. 調 査 内 容

本研究では，実用的なコードクローン検出ツールである CCFinder から得られるコードクローンについて，コードクローンであるコード片間における識別子の差異を調査する．これによって，コードクローンのうちリファクタリングが容易なクローンの比率を調査し，コードクローンに対するリファクタリング手法の適用可能な範囲を明らかにする.

3.1 調 査 対 象

Apache Commons および SourceForge.net から収集した，Java で記述された 2,142 個のオープンソースソフトウェアについて，コードクローンの識別子情報を調査した.

各ソフトウェアの Java ソースファイルからは，ツールによって自動生成されたコードを事前に取り除いた．これは，自動生成されたコードが手作業で修正されることはなく，また，自動生成されたコード同士が互いにクローンとして検出されることが多いためである.

検出されたクローンの中からは，意味のないコードクローンの除去を行っている．CCFinder は，検出したコードクローンに関して，その中に含まれるトークンの「繰り返し」の除去を行い，残ったトークン数を返す．この値は，たとえば単純な `if` 文の繰り返しや変数宣言の並びのように，あるトークンの部分列が繰り返されている場合に小さくなる．この値が小さ

なコードクローンは，開発者にとって意味のあるコードクローンではないことが多いことが経験的にわかっているため，コードクローンとなっているコード片の 50%以上が CCFinder によって繰り返しとして認識されている場合，調査対象から除外した．

3.2 コードクローン情報に関する前提条件

本研究では，CCFinder が出力するクローンを調査する．CCFinder が検出するクローンはタイプ 1 もしくはタイプ 2 のクローンであるが，どちらのタイプであるかは判定されていない．以下に CCFinder から得られる情報についての前提条件を示す．CCFinder が検出するものと同種のクローン，すなわちタイプ 1 およびタイプ 2 のクローンを検出するツールであり，かつ，下記の前提条件を満たすことができるツールであれば，同等の調査が可能である．

- コードクローン情報は，**クローンセット** *(Clone Set)* と呼ばれる単位の列で表される．各クローンセットは，互いに類似していると CCFinder が判定したコード片の集合である．
- クローンセットに含まれる各コード片はソースコード上で連続である．各コード片は，ファイル名，開始行番号，開始桁位置，終了行番号，終了桁位置の組によって指定される．
- クローンとなっているコード片は,「パラメータ化」という正規化操作を適用すると，同一の長さのトークン列となる．ここでのパラメータ化とは，Java などのプログラミング言語の文法において，識別子およびリテラル，すなわち，開発者が自由に定義することができる変数名や型名，値の出現に対して，それらをすべて無名のトークンに置換する操作を意味する．たとえば，２つの代入文 `i = i + 1 ;` と `z = x + y ;` が与えられたとすると，これらをパラメータ化した結果は，同一の式`$p = $p + $p ;` となる．つまり，これら２つの代入文は，タイプ 2 のクローンであるといえる．

3.3 コード片の差異

CCFinder が抽出するコードクローンは，識別子およびリテラルをパラメータ化した状態で完全一致する文字列であるから，識別子を他の識別子に置換することで，あるコード片から，互いにコードクローンであるような他のコード片に変換することができる．

置換とは，識別子に対する機械的な置き換え処理を意味する．あるコード片 c が与えられたとき，c に含まれる識別子またはリテラル i_1 のすべての出現を i_2 に置き換える処理を $R_{i_1 \to i_2}(c)$ と記述する．たとえば，$c =$ { `x = Math.max(x, y);` } とすると，$R_{x \to z}(c) =$ {`z = Math.max(z, y);`} となる．複数の置換は同時に適用されるものと考える．たとえば，$R_{x \to y, y \to x}(c) =$ {`y = Math.max(y, x);`} である．クローンセット $C = \{c_1, c_2, \cdots, c_k\}$ がどれだけ異なっているかは，これらのコード片を互いに変換するために必要な置換の数によって定義することができる．

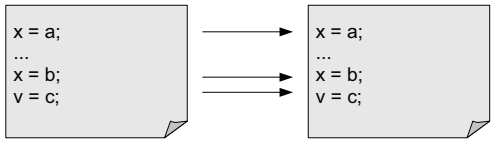

図 4 完全一致
Fig. 4 Exact match

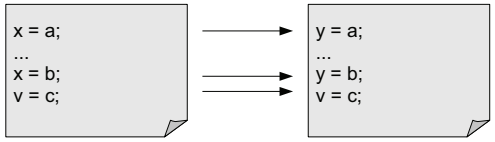

図 5 1 対 1 対応
Fig. 5 1 to 1 mapping

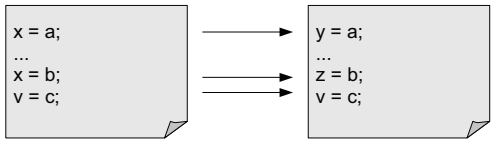

図 6 N 対 N 対応
Fig. 6 N to N mapping

・完全一致

クローンセット C に含まれる任意のコード片の組 c_i, c_j が置換なしで完全一致するとき，このクローンセットに含まれるコード片は完全一致である．例を**図 4** に示す．

・1 対 1 対応

クローンセット C に含まれる任意のコード片の組 c_i, c_j について，$R_1(c_i) = c_j$，$R_2(c_j) = c_i$ となる置換 R_1，R_2 を定義することが可能であるとき，クローンセットに含まれるコード片は，識別子が「1 対 1 で対応する」状態にある．**図 5** に示す例

では，左側のコードに x → y という置換を適用すれば右側のコードを得ることができ，また，右側のコードに y → x という 1 対 1 置換を適用すれば左側のコードを得ることができる．

・N 対 N 対応

クローンセット C に含まれる少なくとも 1 つのコード片 c_i を c_j に変換するための 1 対 1 での置換が定義できないとき，これらは「N 対 N で対応する」クローンである．**図 6** に例を示す．このコードでは，左側のコードに出現する変数 x に対して，1 つの出現を y に，1 つの出現を z に置換しない限り，それを 2 つの変数 y と z に分解することはできない．なお，本研究では，このように 1 対 1 での置換が行うことができないものについては，1 対 N という対応関係になっているものであっても，すべて N 対 N とみなす．

識別子が完全一致，あるいは 1 対 1 で対応していれば，名前の対応関係を適切なリファクタリングによって吸収できる可能性がある．たとえば，変数名であれば 1 つの引数名に統一する，型名であれば総称型を用いる，などの操作が可能である．N 対 N の対応関係の場合は，変数などの値の意味が異なる可能性が高く, 変換が容易なものである可能性は低い．

3.4 識別子の分類

識別子とは開発者が名前を定義することのできる様々な要素のことである．本研究では，プログラミング言語 Java の文法をもとに，識別子の種類を以下のように分類した．

- 変数名．
- メソッド名．
- メソッド名の呼び出しオブジェクト指定子（レシーバ）．ここには変数名，予約語 this，super，任意のクラス名がピリオド（“.”）で区切られて出現する．本研究では，ピリオドで連結された式を 1 つの識別子とみなす．
- 型名．
- リテラル．
- その他の識別子．enum 宣言，アノテーション名，import 文，パッケージ名がここに含まれる．

4. 分析ツールの実装

コードクローン間の識別子情報を得るための解析ツールを作成した．その内容について説明する．

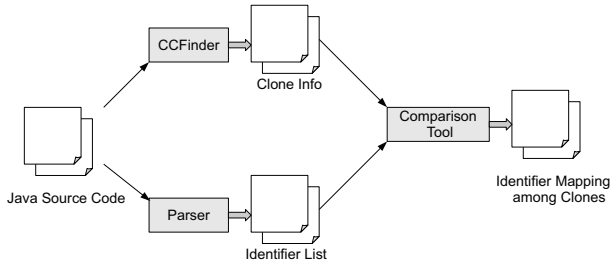

図 7 ツールの動作概要
Fig. 7 An overview of our tool

4.1 ツール概要

図 7 のように，対象となるソースファイルから識別子のリストとコードクローン情報を作り，それを比較することでクローン間の識別子情報を出力する．対象言語は Java に限定しているが，今回の調査方法は，識別子のリスト作成部をプログラミング言語固有の処理に差し替えることで，適用することが可能である．

4.2 識別子リストの作成

識別子リストを作成するためにソースファイルの構文解析を行う必要がある．これには ANTLR を利用することとした．ANTLR の文法ファイルで Java の構文解析を行うものが既に配布されている[4] ので，この文法ファイルに識別子の名前，出現位置，トークン番号をファイルに書き出すような記述を書き加えた．

4.3 識別子の比較

CCFinder によって検出されたクローンセットに対応する識別子リストを取り出し，相互に比較を行うことで，クローン間の識別子の対応関係を計算する．計算の手順は，次の通りである．

(1) コードクローン情報からクローンの位置に対応するファイル名，行番号，桁位置を取得する．
(2) 該当ファイルの識別子リストから，クローンになっているコード位置に含まれる識別子列を抜き出す．
(3) 抜き出した識別子列を, 頭から順に比較する．

なお，この実験ではクローンセット単位での調査を行っているため，3 つ以上のコード片に対する比較が行われることがある．その場合,「すべてのコード片において名前が共通」の場合のみ「一致」として扱い，その他の場合「不一致」として扱う．たとえば，4 つのコード片からなるクローンセットで，ある位置の識別子の名前が a,a,a,b となっていた場合，この識別子は「不一致」としてカウントする．

例として，次の 2 行の文について，1 行ずつをそれぞれクローンのコード片と見たときの比較例を示す．

コード片 1：`for (int i=0; i<10; ++i) sum = calculater.get(base, a[i]);`
コード片 2：`for (int idx=0; idx<20; ++idx) m = Math.max(m, s[idx]);`

まず，これらの各行から，識別子のリストを抽出する．この結果を**表 1** に示す．得られた表から，識別子の対応関係を計算する．この処理は，以下の 2 つのルールによって，識別子をグループ化することで得られる．

- 同じ位置にある各識別子は，同一グループである．
- 1 つのコード片に出現する同名の識別子は，同一グループである．

このルールを表 1 の識別子に適用すると，**表 2** が得られる．この表では，グループの結果が分かりやすくなるように，表 1 の行を入れ替え，また，同一グループになった行間の罫線を取り除く形で示している．この識別子の対応関係を，すべてのクローンセットに関して計算する．

表 1 識別子リストの例
Table 1 Example of Identifier List

コード片 1 の識別子	コード片 2 の識別子	識別子の種類
int	int	型名
i	idx	変数名
0	0	リテラル
i	idx	変数名
10	20	リテラル
i	idx	変数名
sum	m	変数名
calculater	Math	レシーバ
get	max	メソッド名
base	m	変数名
a	s	変数名
i	idx	変数名

各クローンセットに対して識別子の対応関係が計算されたところで，クローンセットに対する分類を計算する．ここでの分類とは，変数名，メソッド名，型名，レシーバ，リテラルの 5 種類について，完全一致でない識別子の組が 1 つ以上あったならば，その識別子は不一致であると考える．5 種類の不一致の有無に基づき，$2^5 = 32$ 個のグループに分類する．

5. 調 査 結 果

Apache Commons および SourceForge.net から収集した 2,142 個のオープンソースソフトウェアに対し，分析ツールを実行し，その結果を集計した．

5.1 識別子の対応関係に基づく分析

識別子の対応関係についての調査結果を**表 3** に示す．この表は，変数名，メソッド名，型名について，クローンセット内で完全一致した識別子の数，1 対 1 の対応が取れた識別子の

表 2 識別子の対応関係の計算結果
Table 2 Result of Calculating Mapping of Identifier

コード片 1 の識別子	コード片 2 の識別子	識別子の種類	対応関係
int	int	型名	完全一致
i	idx	変数名	1 対 1
i	idx	変数名	
i	idx	変数名	
i	idx	変数名	
0	0	リテラル	完全一致
10	20	リテラル	1 対 1
sum	m	変数名	N 対 N
base	m	変数名	
calculater	Math	レシーバ	1 対 1
get	max	メソッド名	1 対 1
a	s	変数名	1 対 1

表 3 識別子の対応関係
Table 3 Mapping of Identifier

分類	完全一致	1 対 1	N 対 N	合計
変数名	558,914	251,615	42,744	853,273
メソッド名	661,002	248,929	79,587	989,518
型名	625,483	134,951	36,172	796,606
合計	1,845,399	635,495	158,503	2,639,397

組の数，N 対 N の対応が取れた識別子の組の数を数え上げたものである．たとえば，図 4 に登場したコード片の組では，変数 x, a, b, c, v という 5 つの変数がそれぞれ完全一致で対応していることから，変数名の完全一致の項目に 5 が加えられる．また，図 6 のコード片の組では，左側のコードの x に対応する y, z の変数を合わせて 1 組と数え，N 対 N に 1 を加算する．この集計結果から，N 対 N に対応する識別子は，変数名のうち 5%，型名とメソッド名を合わせても 6%にすぎない．つまり，識別子が 1 対 1 に対応することを，リファクタリング手法において仮定することが可能である．このことは，多くのコード片において，変数名を引数などで単純に置換できるほか，総称型の導入のような操作を行いやすいことを示している．

5.2 識別子の分類に基づく分析

表 4，**表 5** に，クローンセットの分類結果を示す．先頭の列である分類名は，以降の説明に用いるラベルである．変数名，メソッド名，型名，レシーバ，リテラルの各列は，クローンセットの分類条件である．たとえば，“x” が書かれている行は，そこに含まれるすべてのクローンセットにおいて，コード片の該当要素が異なっていたことを示す．たとえば，ラベル VR の行は，変数名とレシーバがそれぞれ少なくとも 1 つは異なっているようなクローンセットの個数である．

この表では，メソッド名および型名に変更が加わっていないクローンセットに対してのみラベルを割り当てている．完全一致（EXACT）および変数名のみ (V) の変更に該当するクローンセットは 183,918 個であり，全体の約 26.5 %に相当する．これらのコードクローンは，リファクタリングが容易なクローンであることが期待される．また，メソッド名と型名の変更を持たないクローンセットの集合は 305,291 個であり，全体の約 43.9 %に相当する．レシーバやリテラルの変更については，必ずしも変更が容易とは限らないこともあるが，メソッド抽出リファクタリングにおいては，引数による共通化が可能な範囲である．残りの約 56.1 %は，メソッド名に対する動的束縛の適用や，総称型の使用など，複雑な手順を含んだリファクタリングが必要であり，実際の適用は困難であると考えられる．

5.3 考　　察

本研究の分析は，Java のオープンソースソフトウェアを対象としたものである．調査したクローンセットの個数がプロジェクトでどの程度偏っているかを調べるために，横軸にプロジェクトをクローンセットの多い順に並べ，縦軸にそのプロジェクトまでのクローンセッ

表 4　クローンセットの分類 (1/2)
Table 4　Classification of Clone Sets(1/2)

分類名	変数名	メソッド名	型名	レシーバ	リテラル	クローン数	比率
EXACT						156,284	22.5%
V	x					27,634	4.0%
		x				24,944	3.6%
			x			14,769	2.1%
R				x		11,184	1.6%
L					x	40,544	5.8%
	x	x				20,815	3.0%
	x		x			10,348	1.5%
	x			x		20,743	3.0%
VL	x				x	36,826	5.3%
		x	x			9,257	1.3%
		x		x		9,864	1.4%
		x			x	37,022	5.3%
			x	x		1,973	0.2%
			x		x	7,157	1.0%
RL				x	x	6,198	0.9%
	x	x	x			11,751	1.7%
	x	x		x		24,216	3.5%

表 5　クローンセットの分類 (2/2)
Table 5　Classification of Clone Sets(2/2)

分類名	変数名	メソッド名	型名	レシーバ	リテラル	クローン数	比率
	x	x			x	26,166	3.8%
	x		x	x		9,711	1.4%
	x		x		x	9,454	1.4%
VRL	x			x	x	26,621	3.8%
		x	x	x		3,038	0.4%
		x	x		x	7,052	1.0%
		x		x	x	14,639	2.1%
			x	x	x	1,246	0.2%
	x	x	x	x		26,592	3.8%
	x	x	x		x	9,228	1.3%
	x	x		x	x	40,622	5.8%
	x		x	x	x	9,598	1.4%
		x	x	x	x	3,045	0.4%
	x	x	x	x	x	25,672	3.7%
その他						11,271	1.6%
合計						695,484	1.00

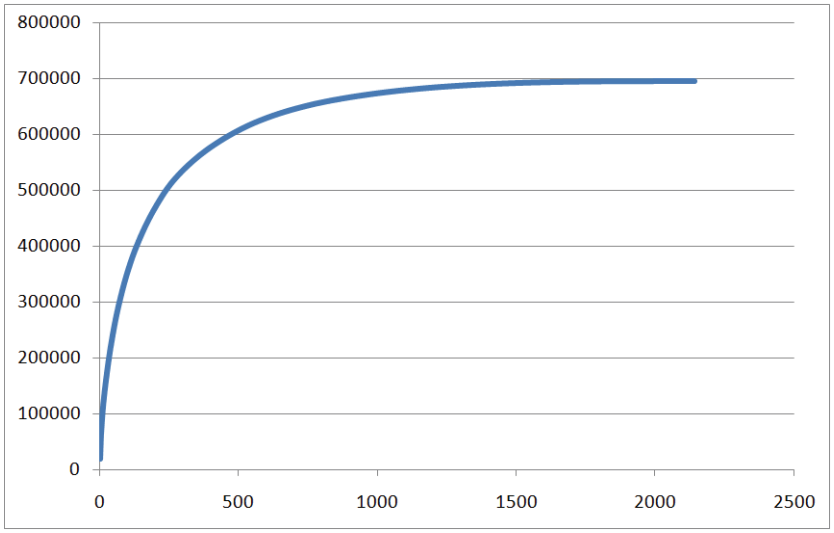

図 8 各プロジェクトにおけるクローンセット数の分布
Fig. 8 Distribution of the Number of Clone Sets in Projects

トの合計数を計算した結果を**図 8** に示す．上位 100 件のプロジェクトで過半数の 355,265 個のクローンセットを持っており，上位 272 件のプロジェクトで 75 %，600 件で 90 %を占める．これは調査対象プロジェクトのおよそ 30 %であり，本研究の調査は，これら上位のプロジェクトの特徴を強く反映したものとなっている．

本研究では，約 4 分の 1 のコードクローンが完全一致または変数名の 1 対 1 対応であり，集約が期待されるコードクローンであるという結果が出たが，実際にそれらのクローンセットに対してリファクタリングを施すべきかどうかはまた別の話になる．集約を行うことによって役割が不明瞭になるなど逆にコードの把握が難しくなってしまうケースも考えられ，これはリファクタリングの理念に反する．本研究で作成したツールから得られる，リファクタリングが容易であるか，容易でないかという指標に対して，開発者のリファクタリングすべきか，すべきでないか，という判断結果を比較する必要がある．

6. ま と め

本研究では，コードクローン間の識別子名の変更内容を分析することで，リファクタリング手法の適用容易性について調査を行った．具体的な調査手順としては，識別子を変数名やメソッド名などの種別によって分類し，どの種類の識別子名が変更されたか，という観点からコードクローンを分類した．

調査の結果，検出されたクローンセットのうちリファクタリングが容易なものは約 4 分の 1 というデータが得られた．また，変数名などの識別子は，95 %が完全一致あるいはコード片の間で 1 対 1 の対応関係を取ることができることが明らかになった．

今後の課題としては，本研究で作成したツールによって得られるリファクタリングが容易であるか，容易でないかという指標に対して，開発者のリファクタリングすべきか，すべきでないか，という判断結果を比較することが挙げられる．また，変数などが対応関係を持つことを前提としたリファクタリングの自動化支援が挙げられる．

謝辞 本研究は科研費（23650015）の助成を受けたものである．

参 考 文 献

1) Balazinska,M., Merlo,E., Dagenais,M., Laguë,B. and Kontogiannis,K. Measuring clone based reengineering opportunities. *Proc. the 6th International Software Metrics Symposium* (METRICS 1999), pp. 292–303, Boca Raton, FL, USA, November (1999).
2) Bellon,S., Koschke,R., Antoniol,G., Krinke,J. and Merlo,E. *Comparison and Evaluation of Clone Detection Tools.* *IEEE Transaction on Software Engineering*, Vol.33, No.9, pp. 577–591 (2007).
3) Fowler,M. *Refactoring: Improving the Design of Existing Code.* Addison Wesley (2000).
4) ANTLR Parser Generator. `http://www.antlr.org/`.
5) 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. 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. 電子情報通信学会論文誌, Vol. J88-D-I, No.2, pp. 186–195 (2005).
6) Kim,M., Bergman,L., Lau,T. and Notkin,D. An ethnographic study of copy and paste programming practices in OOPL. *Proc. the ACM-IEEE International Symposium on Empirical Software Engineering* (ISESE 2004), pp. 83–92, CA, USA, (2004).
7) Lague,B., Proulx,D., Mayrand,J., Merlo,M.E. and Hudepohl,J. Assessing the benefits of incorporating function clone detection in a development process. *Proc. the 13th International Conference on Software Maintenance* (ICSM 1997), pp. 314–321, Bari, Italy, (1997).