特別研究報告

題目

リファクタリング支援のための
コードクローンに含まれる識別子の変更内容分析

指導教員

井上 克郎 教授

報告者

工藤 良介

平成 23 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成22年度 特別研究報告

リファクタリング支援のためのコードクローンに含まれる識別子の変更内容分析

工藤 良介

内容梗概

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）に含まれる，あるクローンにバグが存在した場合，そのクローンセットに含まれるクローン全てについてバグの修正が必要かどうかを検討しなければならない．そのため，コードクローンの存在はソフトウェアの保守のコストを増大させてしまう．よって，可能ならばソフトウェア中のコードクローンを解消することが望まれる．

コードクローンを解消する手法の１つとして，リファクタリングを用いた手法が提案されている．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．コードクローンに対するリファクタリングでは，クローンを集約し，1つのメソッドとして独立させる．クローンセットに含まれるクローン間の違いを吸収し，1つのメソッドにすることで修正が容易になり，保守コストを下げることができる．しかし，クローンセットに含まれたコード片の差異によって，集約の難易度は大きく異なる．例えば，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．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，集約が困難になる．過去の研究では，コードクローンの検出や集約の方法そのものについては数多く提案がなされているが，集約の期待できるクローンセットと集約が困難なクローンセットの割合については，調査が行われていなかったため，実際に多くのクローンセットに対してリファクタリングが適用可能であるのかは判明していなかった．

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．

調査の結果，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変更のみであり，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できることが明らかになった.

主な用語

コードクローン
リファクタリング
識別子

目 次

1 まえがき **4**

2 背景 **6**

2.1 コードクローン 6

2.2 リファクタリング 8

2.2.1 コードクローンのリファクタリング 9

2.2.2 集約することが困難なコードクローン 9

3 調査内容 **13**

3.1 調査対象 13

3.2 コードクローン情報に関する前提条件 13

3.3 コード片の差異 14

3.4 識別子の分類 16

4 分析ツールの実装 **18**

4.1 ツール概要 18

4.2 識別子リストの作成 18

4.3 識別子の比較 19

4.4 出力 20

5 調査結果 **23**

5.1 識別子の対応関係に基づく分析 23

5.2 識別子の分類に基づく分析 23

5.3 考察 25

6 まとめ **27**

謝辞 **28**

参考文献 **29**

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）に含まれる，ある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 [10]．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，特に大規模なプロジェクトにおいては，開発者の大きな負担となっている．

コードクローンを内包した状態でのコード修正には，クローンの修正漏れの可能性があるため，長期的に保守が計画されているソフトウェアでは，コードクローンを解消することが強く望まれる．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．これにより，不具合が発見されたとしても一箇所の修正で済ませられるため，保守コストを抑えることができる．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．例えば，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期待できる．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．このような集約を正しく実行することは困難である．過去の研究では，コードクローンの検出や集約の方法そのものについては数多く提案がなされているが，集約の期待できるクローンセットと集約が困難なクローンセットの割合については，調査が行われていなかったため，実際に多くのクローンセットに対してリファクタリングが適用可能であるのかは判明していなかった．

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．

具体的な調査の手順としては，コードクローン情報によって対応付けられたソースコードを取り出し，識別子の位置に基づく対応関係を計算する．識別子の対応関係は，コードクローンになっている１つのコード片から他のコード片へと変換するような，識別子の名前の

置換によって表現した．コードクローンの定義には数種類あるが，識別子名の違いについて調査するという手法の関係上，クローン間で識別子以外の部分に違いがあると都合が悪い．そのため，本研究では既存のツール CCFinder[7] によって検出されるパラメタライズドクローン，すなわち，互いに同じ長さのコード片であり，変数名や型名などの開発者が決定する識別子名だけが異なるようなコード片だけを調査対象とした．コードの挿入や削除によって不一致部分が生じた物，いわゆるギャップを含むものは対象外としている．

オープンソースソフトウェア 2142 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．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変更のみであり，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できることが明らかになった．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変更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．

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，各コードクローン検出ツールの紹介とその検出のためのアプローチについて述べる．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 [9]．しかし，ソースコードの再利用においては，複製したコード片をそのままの形で，複製先で利用できるとは限らないため，再利用する環境に合わせて複製したコード片を適切に変更する必要がある．そのため，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を除き，完全に一致するコードクローン (図 1)

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 (図 2)

タイプ 3

タイプ 2 のおける変更に加えて，一部に文の挿入や削除が行われたコードクローン (図 3)

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，バグを含んだコード片が複製されてしまった場合，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，長期的に保守する計画

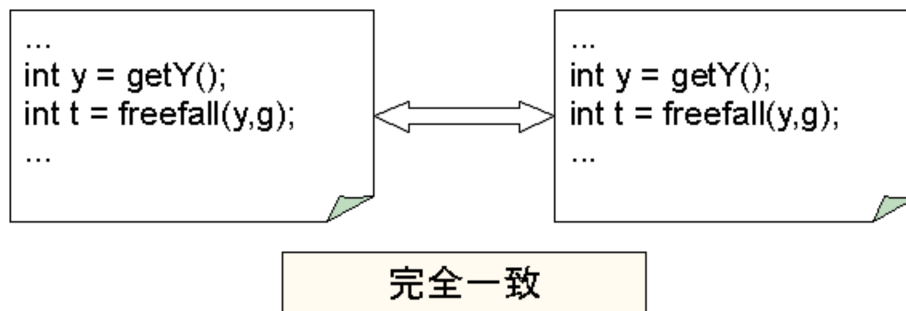

図 1: タイプ 1 のコードクローン例

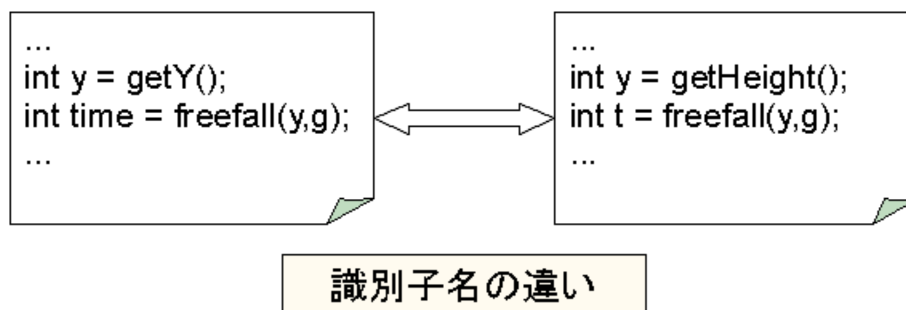

図 2: タイプ 2 のコードクローン例

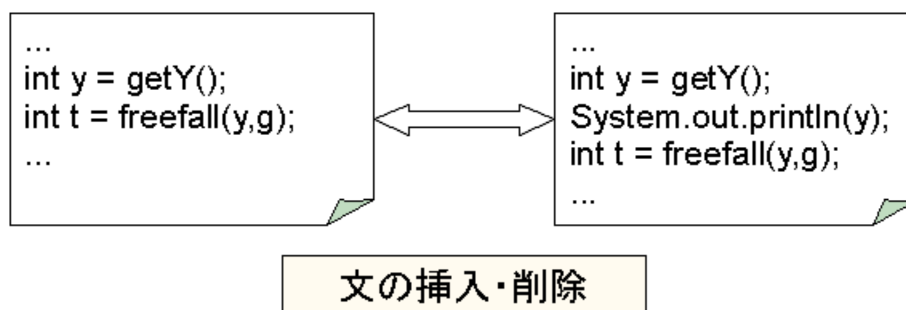

図 3: タイプ 3 のコードクローン例

のあるソフトウェアをリリースする前に，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．

コードクローンを自動的に検出するためのツールは，数多く提案されている．コードクローンの検出とはソースコードの比較であり，行，字句解析におけるトークン，抽象構文木，プログラム依存グラフなど，様々な比較の基準が用いられている．

これらの検出ツールのうち，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 は，与えられたソースファイル集合からコードクローンを検出して，コードクローンの位置情報を出力する．また，CCFinder は 100 万行規模の大規模なソースコードに対して，数分から数時間でコードクローンを検出できるほか，C/C++, Java，COBOL，Fortran など多言語に対応していることが特徴である．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．

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 [4]．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．

リファクタリングの具体的な手法として様々なものが提案されている [4, 8]． リファクタリング手法の例を次に示す．

メソッドの抽出

1 つのメソッドが長すぎてメソッドの処理内容を理解することが容易でない場合に，メソッド中から 1 つの意味を持つ処理を抽出し，新たなメソッドとして独立させる．

メンバの移動

クラス中にそのクラスの役割に合致しないメンバが含まれる場合，メンバを適切なクラスに移動することで，それぞれのクラスの役割を明確にする．

クラス・メソッド・属性の名称を変更

クラス，メソッド，属性の名称が不適切な場合に，その役割を正しく表す名前に変更

する．

これらのリファクタリング手法を用いることで，ソースコードの可読性が向上したり，開発者がソースコードを間違って理解することが少なくなり，最終的にはソフトウェアの保守コストを削減することにつながる．

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 [6]．

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 4 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．

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．

また，コード片の間にある差異をメソッド呼び出しとして吸収できないようなコードクローンは，前述の手法が適用できない．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

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．(図 5)

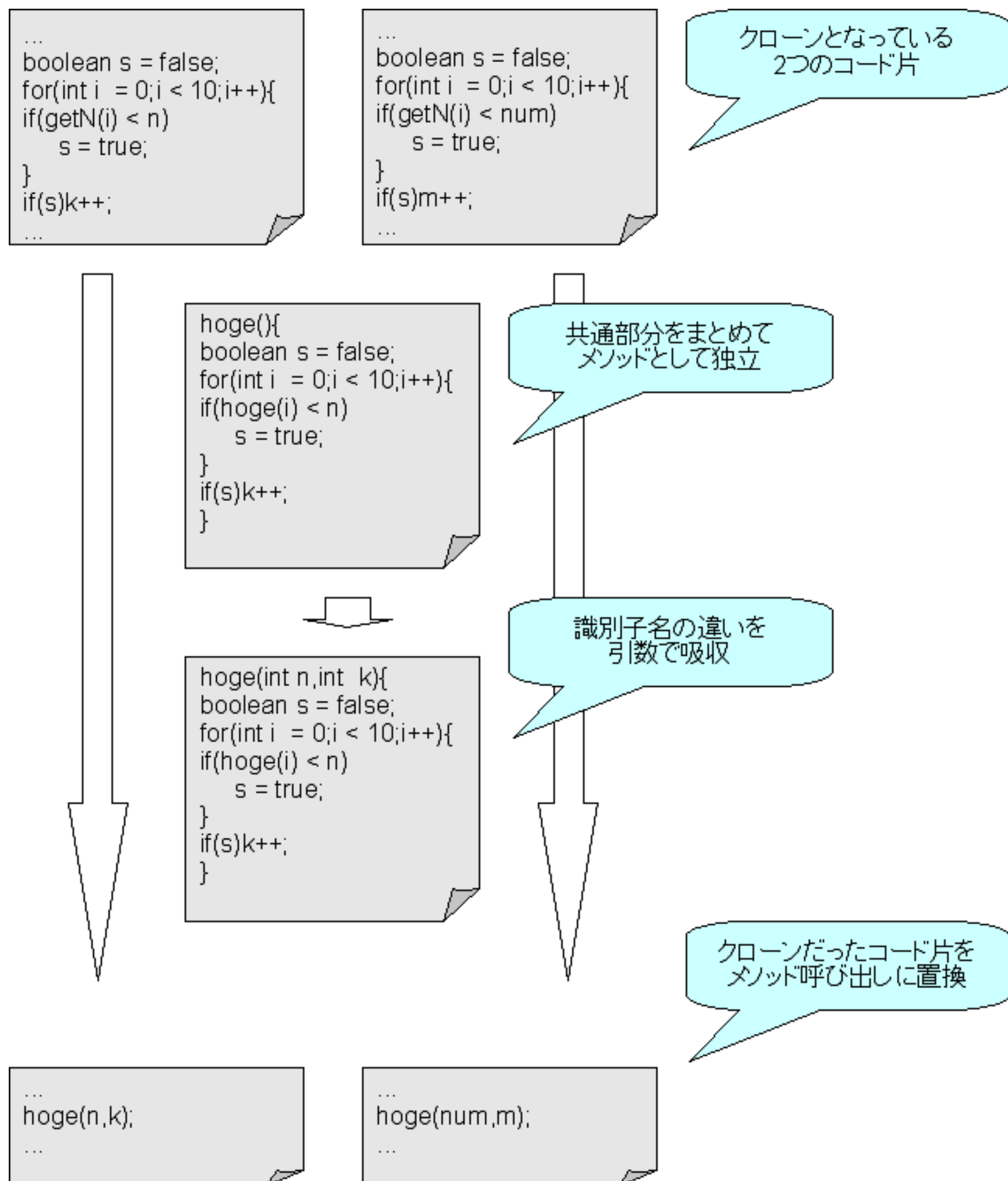

図 4: コードクローンを解消するためのリファクタリング例

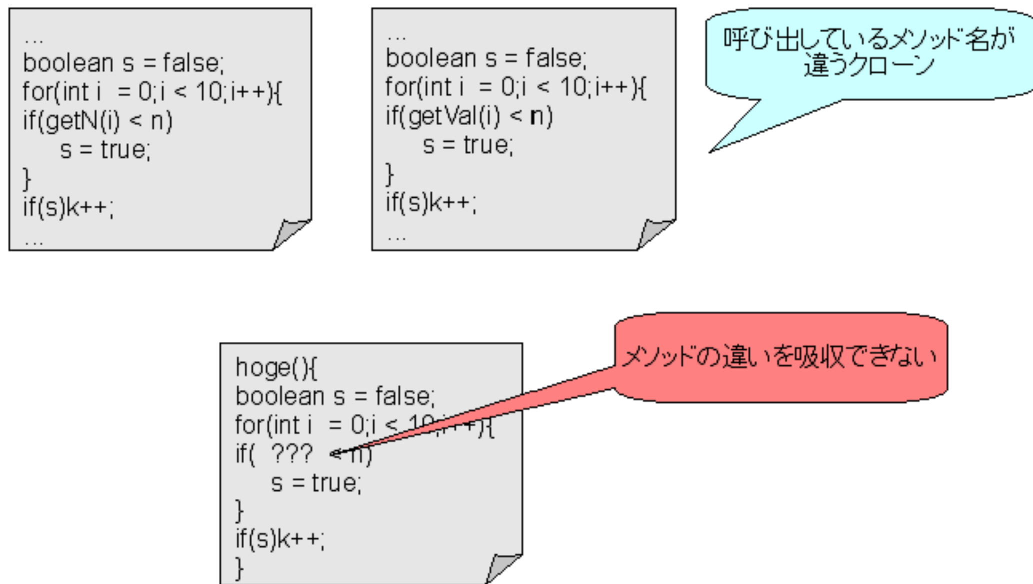

図 5: メソッド名が異なる場合

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．(図 6)

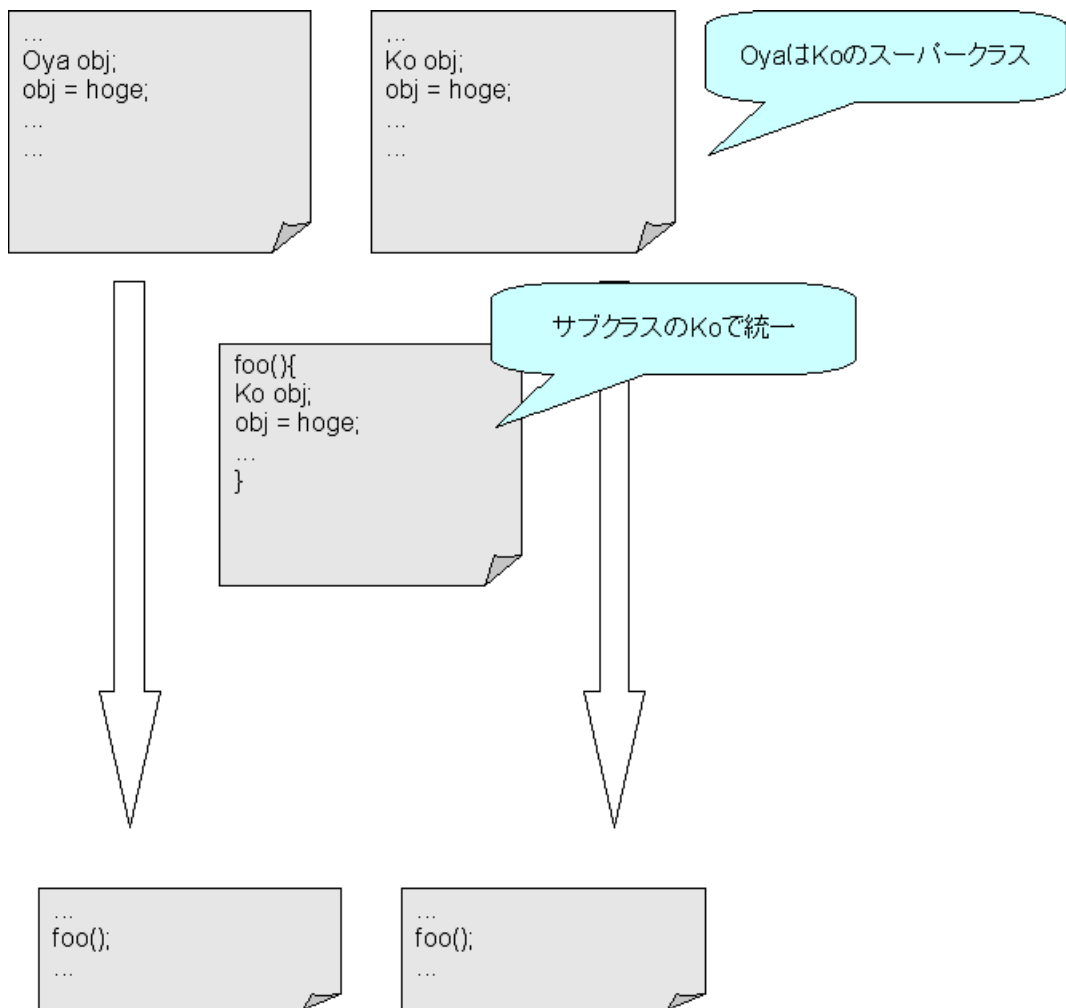

図 6: 型名が異なる場合

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の間における差異の量を調査する．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明らかにし，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[3] および SourceForge.net[12] 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を対象とした．これらのソフトウェアについて，それぞれ個別に CCFinder を実行し，コードクローン情報を抽出しておくものとする．

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．コードの自動生成ツールの１つである ANTLR [5] は，生成したソースファイルの先頭１行目に，“`// ANTLR`” から始まるコメントを埋め込むことから，このコメントが存在するファイルを対象から除外した．

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ある．各

クローンセットは，互いに類似したと 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（Identifier）およびリテラル（Literal），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

コード片の差異を表現する方法には様々なものがあり，たとえば，コードの行単位で表現するもの（いわゆる UNIX の `diff` コマンド）や構文木の形状で表現するもの [11]，トークン単位で表現するものが挙げられるが，本研究では，トークン中の識別子単位で表現する．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．

・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のクローンである．例を図 7 に示す．

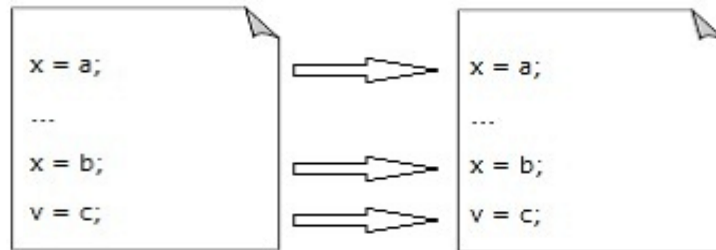

図 7: 完全一致

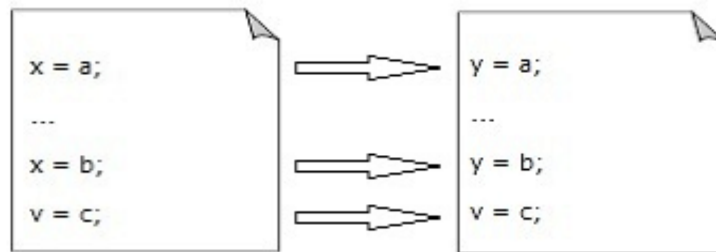

図 8: 1 対 1 対応

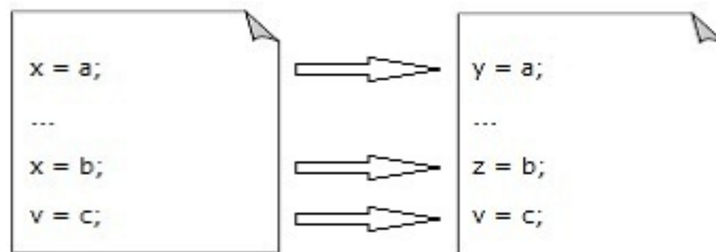

図 9: N 対 N 対応

- **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 8 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 9 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 とみなすことにしている．

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

識別子は開発者が定義することのできる様々な要素の名前である．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．なお，`int` や `long` などの組み込み数値型（いわゆる primitive types）は本来は予約語であって識別子ではないが，本研究では，これらの型名も識別子の一種とみなす．

- リテラル．いわゆる定数値などは，変数名と置き換わることが多いため，リテラル自体は識別子ではないが，識別子の一種とみなす．
- その他の識別子．ここに分類されるのは，リファクタリングなどで重要視されないと推測された識別子である．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．

この分類を用いることで，コード片の間の置換を特徴づけることが可能である．本研究では，クローンセットの「分類」とは，クローンセット内で用いるすべての置換（N 対 N のものを含めて）に出現する識別子の分類の和集合であると考える．たとえば，図 8 の例では，２つのコードを相互変換するために変数名に対する置換を定義すればよいことから，クローンセットが持つ差異は「変数」のみである．図 9 のように N 対 N の置換を行う必要があるものについても，分類結果は同様に「変数」となる．また，図 2 の例では，変数名とメソッド名についてそれぞれ置換を行う必要があることから，クローンセットは「変数, メソッド」という差異を持っていることになる．なお，変数名がリテラルへ置換されるといった異なる種類の識別子への変更は,「種別をまたがる置換」という特殊な分類を行う．

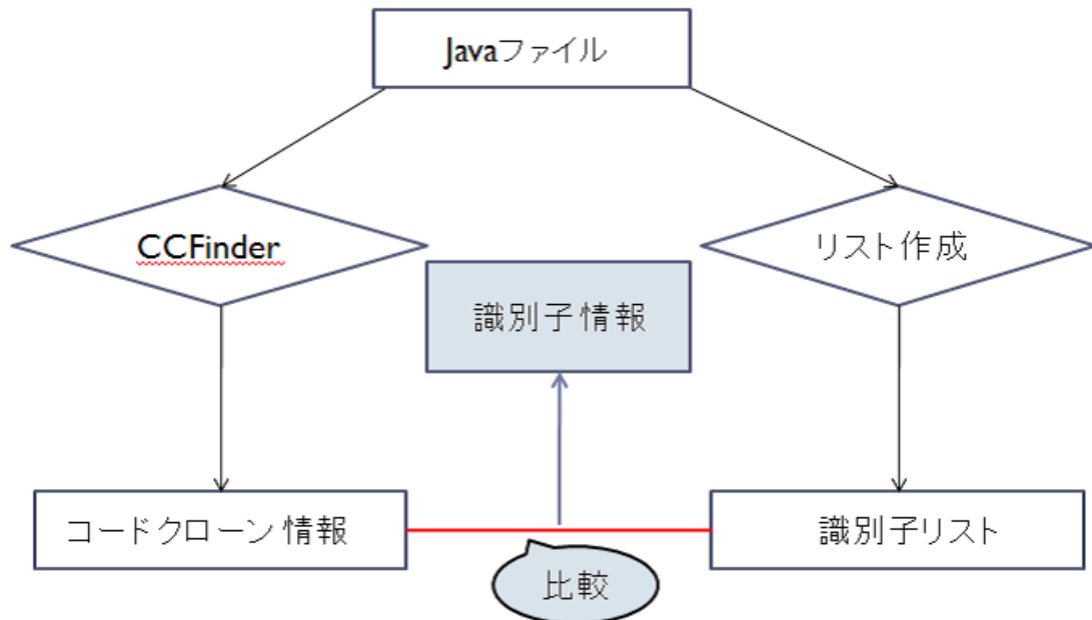

図 10: ツールの動作概要

4　分析ツールの実装

識別子の変更情報を得るために，ソースコードと CCFinder の出力結果を解析するツールを作成した．その内容について説明する．

4.1　ツール概要

図 10 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．コードクローン情報は既存のツール CCFinder によって作られるので，今回作成したのは識別子リストの作成と，識別子リストとコードクローン情報の比較を行う部分である．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 [5] 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，識別子の対応関係を計算する．計算のステップは，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. 識別子名を上から順に比較する．

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「変更なし」として扱い，その他の場合「変更あり」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「変更あり」としてカウントする．

表 1: コードクローンに含まれる識別子リストの例

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

例として，次の 2 行の文について，1 行ずつをそれぞれ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]);`

CCFinder の出力結果から，`for` 文の先頭から行末のセミコロンまでがコードクローンとして提示されたとすると，これらの各行から，予約語や記号部分を取り除いた識別子のリストを抽出する．この結果を表 1 に示す．変数名，型名，レシーバなどの構文情報は，ソースコードの構文解析によって得られる情報である．これらの情報は，ファイルがコンパイル可能であれば（Java の文法を満たしていれば）意味解析を行うことなく抽出することができる．

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．

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，その識別子には変更が加わったと考える．５種類の変更内容それぞれの有無に基づき，$2^5 = 32$ 個のグループに分類する．

4.4　出力

各クローンセットに対して，そのコード片間の差異による分類を行った結果を集計して出力する．以下は，プログラムごと，また全体で集計される最終的な出力ファイルに記述される内容である．

・クローンセットの番号 CCFinder の出力と対応したクローンセットの番号．

- **名前の一致しない識別子の数と全識別子の数** コードクローンとなっているコード片に含まれた識別子の総数を示す.
- **識別子の分類ごとの名前の一致しない識別子の数と全識別子の数** 変数名, メソッド, 型, レシーバ, リテラルの５種, その他の識別子, そして分類間での変更の７通りの識別子の個数を示す.
- **名前の対応関係**
  完全一致, 1 対 1 対応, N 対 N 対応のいずれに該当するかを示す.

各クローンセットについては, 詳細な分析用に, 以下の情報が出力される.

- **クローンセットの番号**
  CCFinder の出力と対応
- **ファイル名とクローンの位置**
- **RNR(非繰り返し率)**
  コードクローンのフィルタリングに使用される指標の１つである.
- **識別子が一致しているかどうか**
  すべてのコード片で識別子が同一であるときのみ真となる.

表 2: 識別子の対応関係の計算結果

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

- **識別子の分類**
- **コード片ごとの識別子名**
- **識別子の対応関係**

  この情報は，表 2 に示したものと同様の内容を，テキスト形式で出力している．

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は，全部で 539,455 ファイルとなった．

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 7 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 9 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 4，5 に，クローンセットの分類結果を示す．1 行ごとにクローンセットの分類と，その分類に所属するクローンセットの数，そしてクローンセット数の全体に対する比率を示している．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変更されていたことを示す．たとえば，ラベル V の行は，変数名の列にだけ “x” が書か

表 3: 識別子の対応関係

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

れているので，変数名だけが変更されたクローンセットの個数を示している．同様に，VR の行は，変数名とレシーバがそれぞれ少なくとも１つは変更され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，適切なクラス継承の導入，総称型の使用

表 4: クローンセットの分類 (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%

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．オープンソースソフトウェアの中では，SourceForge.net で公開されている様々なソフトウェアを含んでおり，一般的な Java プログラミングのやり方が，結果に反映されていることが期待される．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トの合計数を計算した結果を図 11 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．

表 5: クローンセットの分類 (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

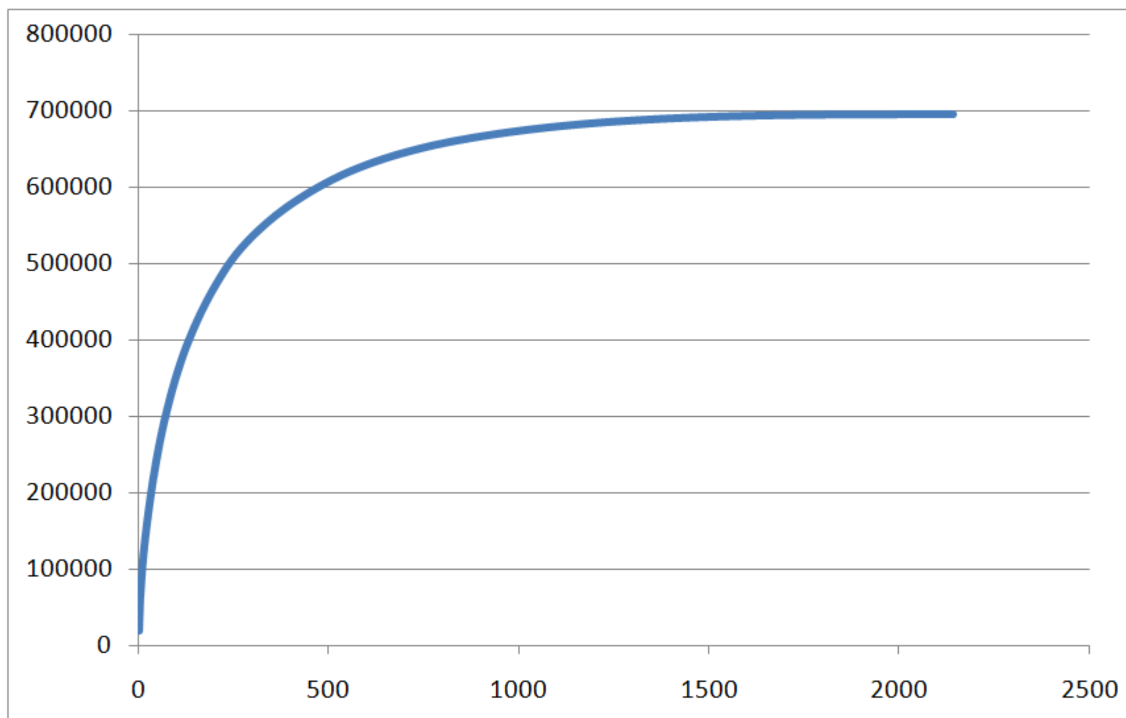

図 11: 各プロジェクトにおけるクローンセット数の分布

コードクローンは，各プロジェクトごとに個別にソースコード集合を分割した状態で検出した．これは，オープンソースソフトウェアでは，プロジェクト間でのソースコードの関係が少ないと考えたためである．企業における開発では，関連したソフトウェアプロダクト間でのソースコードの再利用が行われることもあるため，企業のソースコードに対して，また，プロジェクト間でのクローン検出を適用すると，結果が変わる可能性がある．

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

本研究の全過程を通して，常に適切な御指導及び御助言を賜りまし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 克郎教授に心より深く感謝いたします.

本研究において，随時適切な御指導及び御助言を賜りまし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 松下 誠准教授に深く感謝いたします.

本研究において，終始適切な御指導及び御助言を頂きまし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 石尾 隆助教に深く感謝いたします.

本論文を作成するにあたり, 適切な御指導および御助言を頂きまし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 渡邊 結氏に深く感謝いたします.

本論文を作成するにあたり, 数多くの御指導および御助言を頂きまし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 伊達 浩典氏に深く感謝いたします.

本論文を作成するにあたり, 様々な御指導および御助言を頂きまし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 悦田 翔悟氏, 中野 佑紀氏両名に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂い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] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the 6th International Software Metrics Symposium (METRICS 1999)*, pp. 292–303, Boca Raton, FL, USA, November 1999.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transaction on Software Engineering*, Vol. 33, No. 9, pp. 577–591, September 2007.

[3] Apache Commons. `http://commons.apache.org/`.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 2000.

[5] ANTLR Parser Generator. `http://www.antlr.org/`.

[6] 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, February 2005.

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

[8] J. Kerievsky. *Refactoring to Patterns.* Addison Wesley, 2004.

[9] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004)*, pp. 83–92, CA, USA, August 2004.

[10] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM 1997)*, pp. 314–321, Bari, Italy, October 1997.

[11] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of*

the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 188–197, Chicago Illinois, USA, September 2004.

[12] SourceForge.net. `http://sourceforge.net/`.