# コードクローンの分布情報を用いた特徴抽出手法の提案

服部剛之，肥後芳樹，楠本真二，井上克郎
大阪大学 大学院情報科学研究科
E-mail: thattori@ist.osaka-u.ac.jp

## 1 まえがき

近年，ソフトウェアシステムの大規模化，複雑化に伴い，ソフトウェア保守に要するコストが増加してきている．ソフトウェア保守を困難にしている 1 つの要因として，コードクローンが指摘されている．

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を同一システム中に持つコード片のことである．それらの多くは，既存システムに対する変更や拡張時における“ コピーとペースト ”による安易な再利用の際に発生する．例えば，あるコード片にバグが含まれていた場合，そのコード片に対応する全てのコードクローンについて，修正を行うかどうかを検討する必要がある．また，保守性を高めるため，クローンセット (コードクローンの同値類) を 1 つのサブルーチン等にまとめる方法が考えられる．そのためには，すべてのコードクローンを検出する必要がある．

これまでに様々なコードクローン検出法やツールが提案されている [1][2][3][4][5][6][7][8]．我々の研究グループでもコードクローン検出ツール CCFinder[5] と分析環境 Gemini[11] を開発してきている．ユーザは CCFinder, Gemini を用いることにより，コードクローンの検出・分析，ソースコードの修正を容易に行うことができる [9].

しかし，CCFinder が検出する全てのコードクローンが，ユーザにとって必ずしも有益な情報であるとは限らない．それは，調査の関心がないコードクローン，例えば定型処理のコードクローンが含まれているためである．この問題を解決するためには，コードクローンの特徴を知る必要がある．

そこで，本研究では，Gemini で計測可能なメトリクスを用いてコードクローンの特徴を抽出する手法を提案する．具体的には，コードクローンを含むファイルの数，コードクローンを含むファイル群のディレクトリ階層上での距離，コードクローンに含まれる処理における重複の度合いという，3 つのメトリクスを用いて分類を行う．また，本手法を幾つかのソフトウェアに対して適用実験を行い，その妥当性を評価した．その結果，本手法によって同じ分類に属するコードクローンは，ソフトウェアの記述言語によらず，同様の特徴を持つことを確認した．

ユーザは分類ごとの特徴を用いて，調査の対象としないコードクローンをフィルタリングすることが可能となる．ユーザはフィルタリングを行うことにで，より効率的にコードクローン分析を行うことができると期待される．

以降 2 節では，コードクローン検出ツール CCFinder について説明する．3 節では，メトリクスの値を用いたコードクローンの特徴抽出手法の提案を行う．4 節では，3 節で提案した手法を用いて行った適用実験について説明する．最後に 5 節で本研究のまとめと今後の課題について述べる．

## 2 コードクローン検出ツール CCFinder

### 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 2 つの部分トークン列 $\alpha$, $\beta$ が等価であるとき，$\alpha$ は $\beta$ のクローンであると定義する (その逆もクローンであるという)．また，($\alpha$, $\beta$) を**クローンペア** (図 1 参照) と呼ぶ．$\alpha$, $\beta$ それぞれを真に包含するどのようなトークン列も等価でないとき $\alpha$, $\beta$ を極大クローンという．また，クローンの同値類を**クローンセット** (図 1 参照) と呼び，ソースコード中でのクローンを特にコードクローンと呼ぶ．

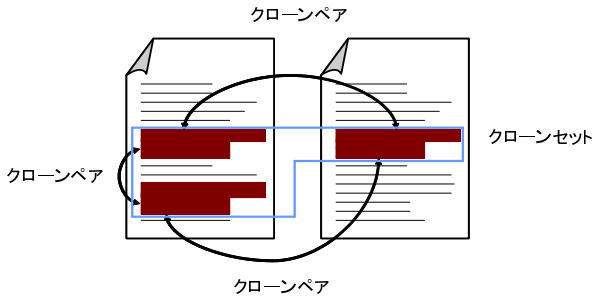


図 1. クローンペアとクローンセット

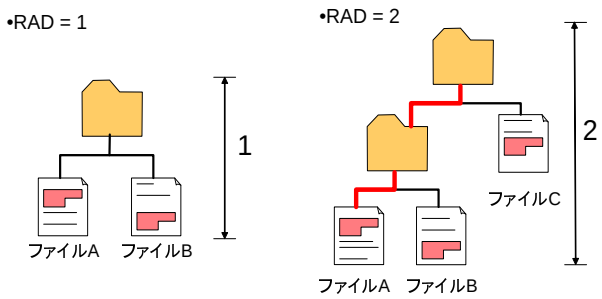


図 2. RAD の値の例

### 2.2 CCFinder

CCFinder は，単一または複数のファイルのソースコード中から全ての極大クローン検出し，それをクローンペアの位置情報として出力する．CCFinder の持つ主な特徴は次の通りである．

**細粒度のコードクローンを検出**

字句解析を行うことにより，トークン単位でのコードクローンを検出する．

**大規模ソフトウェアを実用的な時間とメモリで解析可能**

例えば 10MLOC のソースコードを 68 分[1]で解析可能である [10].

**様々にプログラミング言語に対応可能**

言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C，C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定によるコードクローンは検出可能である．

**実用的に意味の持たないコードクローンを取り除く**

- コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致トークン数を指定することができるため，そのようなコードクローンの検出を防ぐことができる．
- モジュールの区切りを認識する．

**ある程度の違いは吸収可能**

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．
- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違い吸収できる．
- その他，テーブル初期化コード，可視性キーワード (protected，public，private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収できる．

#### 2.2.1 コードクローン検出処理手順

CCFinder のコードクローン検出処理は，以下の 4 ステップで構成されている．

**ステップ 1: 字句解析**

ソースコードをプログラミング言語の文法に沿ってトークン列に変換する．その際，空白とコメントは機能に影響しないので無視される．ファイルが複数の場合には，単一ファイルの解析と同じように処理できるよう，単一のトークン列に連結する．

**ステップ 2: 変換処理**

実用的に意味を持たないコードクローンを取り除くこと，及び，ある程度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，変数名，関数名などは全て同一のユニークなトークンに置換される．

[1]実行環境 Pentium3 650MHz RAM 1GB

**ステップ 3: 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全て検出する．

**ステップ 4: 出力整形処理**
検出されたクローンペアについて，元のソースコード上での位置情報を出力する．

### 2.3 開発現場での CCFinder 利用の問題点

大規模なプログラムのコードクローン分析を行う場合，コードクローンの数は膨大となる．また，調査の関心がないコードクローン (定型処理など) も含まれているため，ユーザが必要な情報を見つけることが非常に困難になる．もし，コードクローンを特徴に応じて分類できるならば，ユーザは必要な情報を得ることが容易になる考えられる．

## 3 分布情報に基づくコードクローン分類手法

### 3.1 概要

本稿では，コードクローンを特徴に応じて分類する手法を提案する．分類を参照することにより，利用者が調査の関心がないコードクローンをフィルタリングすることが可能となり，コードクローン分析の効率化につながる．本提案手法では，コードクローンの特徴を定量的に表すために，Gemini で計測可能なメトリクスを利用する．Gemini は内部的に CCFinder を実行し，CCFinder から得られた解析結果を基に分析環境を提供するシステムである．

### 3.2 利用するメトリクス

本手法で用いるメトリクスは，Gemini で計測可能なメトリクスのうち，RAD(S)，NIF(S) の 2 つである．以下に各メトリクスについての説明を行う．これらのメトリクスはコードクローンの分布の特徴を表している．

**RAD(S)**
クローンセット S の各要素であるコードクローンを含むファイルから共通の親ディレクトリまでの距離の最大値を表す．RAD の値の例を図 2 に示す．ファイル中の色のついた部分はコードクローンを表す．左の図では，どちらのファイルからも共通の親ディレクトリまでの距離は 1 であるため RAD の値は 1 となる．右の図では，共通の親ディレクトリまでの距離で最大値は，ファイル A とファイル B の 2 であるため RAD の値は 2 となる．

**NIF(S)**
クローンセット S に含まれるコードクローンを所有するファイルの数を表す．この値が大きいほど，コードクローンが多くのファイル中に含まれていることを表している．

### 3.3 分類手順

本手法では前節 3.2 で述べたメトリクス (RAD，NIF) の組み合わせにより分類を行う．分類のモデルを図 3 に表す．それぞれの分類の特徴は次のようになる．

- local (RAD 値低，NIF 値低): クローンセットに含まれるコードクローンがディレクトリ階層上近い少数のファイルに存在する．
- horizontal (RAD 値低，NIF 値高): クローンセットに含まれるコードクローンがディレクトリ階層上近い多数のファイルに存在する．
- vertical (RAD 値高，NIF 値低): クローンセットに含まれるコードクローンがディレクトリ階層上遠い少数のファイルに存在する．
- global (RAD 値高，NIF 値高): クローンセットに含まれるコードクローンがディレクトリ階層上遠い多数のファイルに存在する．

## 4 適用実験

### 4.1 実験概要

本節では，前節で述べた提案手法の適用実験について説明する．本実験では Java，C，C++で書かれたオープンソースソフトウェアを対象とした．それぞれのソフトウェアついて，CCFinder を用いてコードクローンを検

出し，提案手法を適用してクローンセットの分類を行った．また，CCFinder が検出する最小のコードクローンサイズは 30 トークンとし，分類ごとの特徴を人の手によって確認した．30 トークンとは，CCFinder におけるデフォルト値である．なお，コードクローンの詳細を確認する際に Gemini を用いた．また，コードクローンの特徴を評価するために次節 4.2.2 節の指標を定義した．実験を行ったオープンソースソフトウェアのデータサイズを表 1 に示す．

### 4.2 準備

#### 4.2.1 調査の際に用いたメトリクス

ここでは，適用実験で各分類に属するクローンセットの特徴を詳しく調査するために用いたメトリクス RNR(S) について説明する．RNR(S) は 3.2 節で説明したメトリクスと同様に Gemini で計測可能なメトリクスである．このメトリクスは，クローンセットに含まれるコードクローンの性質を表している．

**RNR(S)**

クローンセット S に含まれるコードクローン内において，重複していない処理が存在する割合の平均値を表す．コードクローン c 中の総トークン数を $Tokens_{all}(c)$，繰り返し部分のトークン数を $Tokens_{repeated}(c)$ とすると，RNR(S) は式 (1) で表される．コードクローン c はクローンセット S に含まれるとする．RNR(S) の値が小さいと，クローンセット S に含まれるコードクローンは，特定の処理が頻繁に繰り返されていることになる．

$$RNR(S) = 1 - \frac{\sum_{c \in S} Tokens_{repeated}(c)}{\sum_{c \in S} Tokens_{all}(c)} \quad (1)$$

#### 4.2.2 コードクローンの特徴を表す指標

ここでは，コードクローンの特徴を表す指標について説明する．コードクローンが含まれる関数の名前を用いて以下のように定義した．

- same: コードクローンが同じ名前の関数内に存在する．
- similar: コードクローンが名前の似た関数内に存在する．名前の似た関数とは，関数名の一部に共通した単語を持つ関数のことを指す．(例: addConfiguredInputMapper，addConfiguredOutputMapper，addConfiguredErrorMapper)
- different: コードクローンが名前が全く異なる関数内に存在する．

クローンセットが上述の各指標に該当するかを調べることで，分類の特徴を評価する．また，1 つのクローンセットが複数の指標に該当することを許す．複数の指標に該当する例を図 4 に示す．色のついた部分はコードクローンを表している．この例では，あるクローンセット中の複数のコードクローンは same に該当するが，それ以外のコードクローンと比較すると different に該当する場合を表している．

### 4.3 適用結果 (1)

まず始めに各分類に含まれるコードクローンがそれぞれどのような特徴であるか調べた．この実験の調査対象として Ant[2]を用いた．

RAD，NIF の高低は次のように設定した．RAD 高の範囲を上位 2 値，RAD 低の範囲を下位 2 値とした．また，NIF についても同様に NIF 高の範囲を上位 2 値，NIF 低の範囲を下位 2 値とした．分類ごとの特徴を抽出しやすくするために RAD，NIF の範囲をこのように設定した．そして，その条件下でクローンセットを 1 つずつ人の手で調査した．分類ごとのクローンセットの特徴は以下のようになった．

- local: この分類中で RNR の値が低いコードクローンは，単純なロジックのコードクローンであった．この分類中で RNR の値が高いコードクローンは，同じデータ構造を用いて処理を行う関数内に見られた．例えば，コードクローンが GUI 部品の関数内に存在していた．図 5 がソースコードの例である．色のついた部分がコードクローンとなっている箇所

[2]http://ant.apache.org/

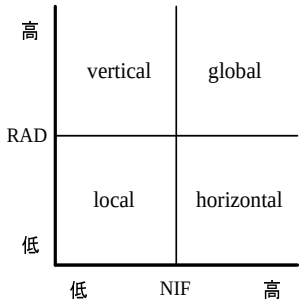


図 3. RAD 値，NIF 値による分類モデル

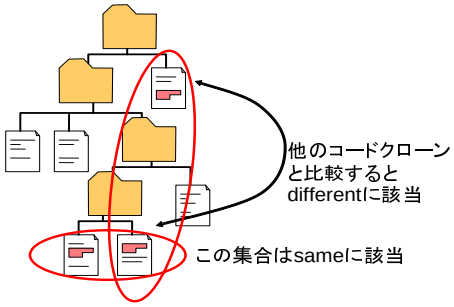


図 4. 複数の指標に該当する例

```
private Button getAboutOkButton() {
  if (iAboutOkButton == null) {
    try {
      iAboutOkButton = new Button();
      iAboutOkButton.setName("AboutOkButton");
      iAboutOkButton.setLabel("OK");
    } catch (Throwable iExc) {
      handleException(iExc);
    }
  }
  return iAboutOkButton;
}
```

図 5. local に属するコードクローンの例

```
} finally {
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException ignore) {
      // ignore
    }
  }
  if (os != null) {
    try {
      os.close();
    } catch (IOException ignore) {
      // ignore
    }
  }
}
```

図 6. global に属するコードクローンの例

である．この例のコードクローンは，ボタンを実装する関数である．

- horizontal: 検出されたコードクローンは，外部プログラムの各機能を呼び出す関数中に見られた．また，コードクローンがそれらのファイルに共通して存在する同名の関数内に見られた．
- vertical: RNR の値が低いコードクローンは，分類 local と同様の結果となった．RNR の値が高いコードクローンは，類似した構造のパッケージ内に見られた．
- global: 検出されたコードクローンは，ファイルの入出力を行っているプログラム中に見られた．内容は入出力ストリームを 2 回続けて閉じるという処理であった．Ant では 2 つのストリームを使うことが多いことから，これらのコードクローンは Ant における定型処理であるといえる．図 6 がソースコードの例である．色のついた部分がコードクローンとなっている箇所である．この例のコードクローンは，2 つのストリームを続けて閉じる処理を行っている．

RNR の値が低い場合，分類 local，vertical の両方に共通した結果が見られた．そのため，異なる分類同士でも，共通した特徴が存在するのではないかと考えた．

### 4.4 適用結果 (2)

#### 4.4.1 分類の汎用性を評価する実験

次に，4.3 節で述べた各分類の特徴が他のソフトウェアについても同様のことが言えるか調べるために，同様の調査を他のソフトウェアにも行うことにした．Ant と同じく Java で書かれた他のソフトウェアを調査し，また Java と異なるプログラミング言語として，C，C++で書かれたソフトウェアも調査することにした．調査対象のソフトウェアは 4.3 節の Ant に加え，オープンソース・ソフトウェア開発サイト SourceForge.net[3]から入手した．

[3]http://sourceforge.net/

Java のプログラム 4 つ，C のプログラム 2 つ，C++のプログラム 2 つについて調査を行った．今回は紙面の都合上，これらのうち Java，C，C++のプログラムそれぞれ 1 つずつについて述べる．その他のソフトウェアに関する詳細は文献 [12] に記述している．

実験条件は次のように設定した．

**実験条件**

- RAD，NIF の高低の閾値を RAD，NIF のそれぞれの最大値の半分とする．
- それぞれの分類において RNR の値によって区分する．これは，4.3 節での RNR の値が低い場合に各分類で共通して見られた特徴を評価するためである．実際の区分の仕方を図 7 に示す．
- 検出されたクローンセット数が多いソフトウェアは，POP[4]の値を一定値以上のクローンセットについて調査する．コードクローン数が多いほうが，コードクローンの特徴を調査しやすいと考えたからである．

調査を行った結果，Java と C，C++のプログラムにおける分類ごとの特徴は似通ったものであった．そのため，プログラミング言語にかかわらず，同じ特徴を持つと考えた．次節 4.4.2 節で評価を行う．

また，RNR の値が低い場合に各分類で共通して見られた特徴，つまりロジックが単純な命令が，クローンセット中に RNR 値の高低によってどの程度見られたかを表 2 に示す．表の値は，クローンセット数 (ロジックが単純な連続した命令に該当する数) を表す．

### 4.4.2 各分類の特徴を評価する実験

次に，分類ごとの特徴を評価するために，4.2.2 節で述べた指標 (same，similar，different) を定義した．これらの指標が関数名に基づいているのは，コードクローンが属する関数名を見ることで，クローンセットに含まれるコードクローン同士の関係をある程度予測できると考えたためである．4.4.1 節で実験を行ったソフトウェアに対して，分類ごとの特徴を指標を用いて評価する実験を 4.4.1 節で述べた実験条件で行った．結果は表 3，表 4，表 5 となった．

表の値は，分類ごとに RNR の値により区分された範囲で，クローンセットがそれぞれの指標に該当した数と，RNR の値により区分された範囲での総クローンセット数に対する割合を表している．

各分類の評価結果は次のようになる．

- local: 多くのクローンセットが same と similar に該当している．同じデータ構造を用いているため，コードクローンとなっている．あるデータ構造の要素に対して処理を行う関数を，処理内容+要素名のような名前で作成したために，similar に該当するコードクローンとなってることが考えられる．
- horizontal: 多くのクローンセットが same に該当している．あるパッケージ内で親クラス下の兄弟クラスが同様の処理を行うため，同じ名前の関数を流用したと考えられる．
- vertical: 多くのクローンセットが different，あるいは same に該当している．different に該当するコードクローンは，類似した構造のパッケージでの異なる処理に存在すると考えられる．また，same に該当するコードクローンは，類似した構造のパッケージでの全く同じ処理に存在すると考えられる．same に該当する場合として，複数のプログラミング言語に対応しているソフトウェアが挙げられる (例 SWIG)．処理対象のプログラミング言語によってパッケージに分け，それぞれのパッケージにおいて同じ処理を行う場合，same に該当するクローンセットが生成される可能性がある．
- global: この分類に属するクローンセットは基本的に different に該当している．定型処理は処理内容にあまり依存しないため，様々な関数内に存在していると考えられる．

また，RNR の値が低いと指標に該当するクローンセットが少ない場合がある．これは，変数宣言やプログラム中で用いる関数の宣言を行っているコード片がコードクローンとなっているからである．このようなコードクローンは，例えばリファクタリングを行う際には重要度が低いと考えられる．

[4] クローンセットに含まれるコードクローンの数

表 1. 調査対象データサイズ

| ソフトウェア名 | プログラミング言語 | クローンセット数 | ファイル数 | コードサイズ |
|---|---|---|---|---|
| Ant | Java | 1,643 | 954 | 約 206,000 行 |
| SWIG | C++ | 2,158 | 208 | 約 85,000 行 |
| Small Device C Compiler | C | 7,550 | 871 | 約 367,000 行 |

表 2. クローンセットの分布 (括弧内はロジックが単純な命令のクローンセット数)

| ソフトウェア名 | | Ant | SWIG | Small Device C Compiler |
|---|---|---|---|---|
| プログラミング言語 | | Java | C++ | C |
| フィルタ | | POP3 以上 | POP3 以上 | POP10 以上 |
| local (RAD 低,NIF 低) | RNR 低 | 94(94) | 237(228) | 162(152) |
| | RNR 中 | 131(80) | 230(142) | 109(94) |
| | RNR 高 | 305(24) | 298(69) | 68(38) |
| horizontal (RAD 低,NIF 高) | RNR 低 | 1(1) | 20(20) | 64(64) |
| | RNR 中 | 1(1) | 8(4) | 0 |
| | RNR 高 | 9(0) | 6(0) | 0 |
| vertical (RAD 高,NIF 低) | RNR 低 | 17(17) | 14(14) | 27(19) |
| | RNR 中 | 12(3) | 13(13) | 5(5) |
| | RNR 高 | 4(0) | 5(1) | 0 |
| global (RAD 高,NIF 高) | RNR 低 | 3(3) | 14(14) | 6(6) |
| | RNR 中 | 0 | 0 | 1(1) |
| | RNR 高 | 4(0) | 0 | 0 |

表 3. Ant の評価結果

| | | Ant | | | |
|---|---|---|---|---|---|
| フィルタ | | POP3 以上 | | | |
| 特徴 | | all | same | similar | different |
| local (RAD 低,NIF 低) 合計 530 | RNR 低 | 94 | 24(25.5 %) | 43(45.7 %) | 11(11.7 %) |
| | RNR 中 | 131 | 47(35.9 %) | 55(42.0 %) | 22(16.8 %) |
| | RNR 高 | 305 | 148(48.5 %) | 159(52.1 %) | 21(6.9 %) |
| horizontal (RAD 低,NIF 高) 合計 11 | RNR 低 | 1 | 1(100.0 %) | 0 | 0 |
| | RNR 中 | 1 | 1(100.0 %) | 0 | 0 |
| | RNR 高 | 9 | 8(88.9 %) | 2(22.2 %) | 2(22.2 %) |
| vertical (RAD 高,NIF 低) 合計 33 | RNR 低 | 17 | 0 | 3(17.6 %) | 14(82.4 %) |
| | RNR 中 | 12 | 0 | 0 | 12(100.0 %) |
| | RNR 高 | 4 | 1(25.0 %) | 1(25.0 %) | 3(75.0 %) |
| global (RAD 高,NIF 高) 合計 7 | RNR 低 | 3 | 0 | 0 | 3(100.0 %) |
| | RNR 中 | 0 | 0 | 0 | 0 |
| | RNR 高 | 4 | 0 | 1(25.0 %) | 3(75.0 %) |

表 4. SWIG の評価結果

| | | SWIG | | | |
|---|---|---|---|---|---|
| フィルタ | | POP3 以上 | | | |
| 特徴 | | all | same | similar | different |
| local (RAD 低,NIF 低) 合計 765 | RNR 低 | 237 | 43(18.1 %) | 57(24.1 %) | 42(17.7 %) |
| | RNR 中 | 230 | 134(58.3 %) | 67(29.1 %) | 14(6.1 %) |
| | RNR 高 | 298 | 215(72.1 %) | 85(28.5 %) | 3(1.0 %) |
| horizontal (RAD 低,NIF 高) 合計 34 | RNR 低 | 20 | 18(90.0 %) | 8(40.0 %) | 3(15.0 %) |
| | RNR 中 | 8 | 6(75.0 %) | 5(62.5 %) | 0 |
| | RNR 高 | 6 | 6(100.0 %) | 0 | 1(16.7 %) |
| vertical (RAD 高,NIF 低) 合計 32 | RNR 低 | 14 | 1(7.1 %) | 4(28.6 %) | 9(64.3 %) |
| | RNR 中 | 13 | 11(84.6 %) | 0 | 0 |
| | RNR 高 | 5 | 5(100.0 %) | 0 | 0 |
| global (RAD 高,NIF 高) 合計 14 | RNR 低 | 14 | 4(28.6 %) | 0 | 12(85.7 %) |
| | RNR 中 | 0 | 0 | 0 | 0 |
| | RNR 高 | 0 | 0 | 0 | 0 |


図 7. RNR の区分

```
ap = (struct area *) new
(sizeof(structarea));
if (areap == NULL) {
      areap = ap;
} else {
      tap = areap;
      while (tap->a_ap)
            tap = tap->a_ap;
      tap->a_ap = ap;
}
ap->a_axp = axp;
axp->a_bap = ap;
```

図 8. 定型処理の例

表 5. Small Device C Compilar の評価結果

| | | Small Device C Complier | | | |
|---|---|---|---|---|---|
| フィルタ | | POP10 以上 | | | |
| 特徴 | | all | same | similar | different |
| local (RAD 低,NIF 低) 合計 339 | RNR 低 | 162 | 44(27.2 %) | 74(45.7 %) | 21(13.0 %) |
| | RNR 中 | 109 | 44(40.4 %) | 73(67.0 %) | 7(6.4 %) |
| | RNR 高 | 68 | 36(52.9 %) | 55(80.9 %) | 8(11.8 %) |
| horizontal (RAD 低,NIF 高) 合計 64 | RNR 低 | 64 | 0 | 0 | 0 |
| | RNR 中 | 0 | 0 | 0 | 0 |
| | RNR 高 | 0 | 0 | 0 | 0 |
| vertical (RAD 高,NIF 低) 合計 32 | RNR 低 | 27 | 5(18.5 %) | 8(29.6 %) | 15(55.6 %) |
| | RNR 中 | 5 | 0 | 1(20.0 %) | 5(100.0 %) |
| | RNR 高 | 0 | 0 | 0 | 0 |
| global (RAD 高,NIF 高) 合計 7 | RNR 低 | 6 | 0 | 1(16.7 %) | 5(83.3 %) |
| | RNR 中 | 1 | 1(100.0 %) | 0 | 1(100.0 %) |
| | RNR 高 | 0 | 0 | 0 | 0 |

### 4.5 考察

異なるソフトウェアでも，分類ごとの特徴は同じであった．このことから，ソフトウェアの記述言語を問わず，メトリクス値に基づいてコードクローンの特徴を抽出することが可能であると言える．

そして，各分類の特徴を利用することで，コードクローンをフィルタすることが可能である．例えば，分類 global に属するコードクローンは定型処理であるため，リファクタリングの対象からはずすことが可能となる．また，分類 vertical に属するコードクローンは，他のパッケージからのアドホックなコピーの恐れがあると考えられ，設計情報の一貫性を確認することが有益ではないかと開発者が判断することが期待できる．

## 5 まとめと今後の課題

本研究では，コードクローンを特徴に応じて自動分類する手法を提案した．具体的には，ファイルのディレクトリ階層上での距離 (RAD)，コードクローンを含むファイルの数 (NIF)，コードクローンに含まれる処理の重複の度合い (RNR)，という 3 つのメトリクスを用いることで分類を行った．

次に，コードクローンの分類を，Java，C，C++の 3 つのプログラミング言語で記述されたオープンソースソフトウェア合計 8 個を対象に行い，分類の特徴の調査を行った．その結果，ソフトウェアの記述言語を問わず，同様の特徴を持つことが判明した．そこで，関数名に着目した指標を用いることで，分類の特徴を評価した．その評価の結果，本手法によって同じ分類に属するコードクローンは，ソフトウェアの記述言語によらず，同様の

特徴を持つことを確認した．

ゆえに，本手法を用いて分類の特徴を用いることで，コードクローンをフィルタリングすることが可能であると言える．そして，本手法により，ユーザが調査の対象としないコードクローンをフィルタリングすることで，より効率的にコードクローン分析を行うことができると期待される．

今回提案した手法では，分類に属するコードクローン数に大きな偏りが見られた．そのため，分類をより詳細にする必要がある．また，ツールとして実装し，ユーザに利用してもらうことで，実際の開発現場での評価を行う必要がある．

## 参考文献


[1] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):pages 1343–1362, 1997.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontoginannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering*, 2000.

[3] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th IEEE International Conference on Software Maintenace-1998*, pages 368–377, 1998.

[4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the 15th IEEE International Conference on Software Maintenance-1999*, pages 109–118, 1999.

[5] 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*, 28(7):pages 654–670, 2002.

[6] R. Komondoor and S. Hirwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[7] J. Krinke. Identifying similar code with program dependece graphs. In *Proceedings of the 8th Working Conference on Reverse Enngineering*, pages 562–584, 2001.

[8] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *resubmitted to Journal of Universal Computer Science*, 2001.

[9] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, pages 327–336, 2002.

[10] 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**, 18(5):pages 47–54, 2001.

[11] 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. **電子情報通信学会論文誌 D-I**, vol.86-D-I, no.12:pages 863–871, 2003.

[12] 服部剛之，肥後芳樹，楠本真二，井上克郎. コードクローンの分布情報を用いた特徴抽出手法の提案. **大阪大学基礎工学部情報科学科 卒業論文**, 2006. `http://sel.ics.es.osaka-u.ac.jp/~lab-db/Bthesis/archive/109/109.pdf`.