特別研究報告

題目

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

指導教員
井上 克郎 教授

報告者
服部 剛之

平成 18 年 2 月 20 日

大阪大学 基礎工学部 情報科学科

平成 17 年度 特別研究報告

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

内容梗概

ソフトウェアの保守作業を困難にする要因の 1 つ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，ソースコード中に同一，または類似したコード片を持つようなコード片のことであり，“ 重複コード ”とも呼ばれている．コードクローンは，既存コードの“ コピーとペースト ”による再利用や，意図的な同一処理の繰り返しなどによりソースコード中に作りこまれる．コードクローンの存在が保守作業を困難にする理由は，修正するコード片のコードクローンが存在した場合，それら全てに対して修正の是非を検討する必要があるからである．そのため，これまでに多くのコードクローン検出・分析手法が開発されている．

我々の研究グループではこれまでにコードクローン検出ツール CCFinder と分析ツール Gemini を開発してきている．CCFinder は大規模なソフトウェアから実用的な時間でコードクローンを検出することが可能である．しかし，CCFinder の出力はテキストベースであり，実システムに対するコードクローン分析に直接 CCFinder の出力を用いるのは効率が悪い．このため，検出したコードクローン分析には Gemini を用いる．Gemini には種々のコードクローン視覚化手法が実装されており，ユーザはコードクローン情報を GUI を通して分析することができる．

CCFinder と Gemini はこれまでに CCFinder/Gemini パッケージとして国内外の個人・組織に配布され，多くのフィードバックが得られた．その中には，コードクローン分析の効率に関するフィードバックが多く含まれていた．CCFinder が検出したコードクローンの中には，ソフトウェア保守作業を妨げるコードクローンのほかに，システムが記述されているプログラミング言語に依存したものや，定型的な処理部分など，ユーザが目で確認を行う必要がないと思われるものが多数含まれている．このようなコードクローンが，調査しなければならないコードクローンと混在しているために，分析作業の効率が悪化するという指摘があった．

そこで，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の提案を行う．具体的には，コードクローンを含むファイルの数，ファイル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の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．

分類の妥当性を評価するために，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，コードクローンの分類を行い，その特徴の調査を行った．

その結果，本手法によって同じカテゴリに分類されたコードクローンは，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．分類ごとの特徴を用いることで，ユーザは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になり，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．

主な用語

コードクローン
ソフトウェア保守
ソフトウェアメトリクス

目 次

1 まえがき **4**

2 準備 **5**

2.1 コードクローンと既存のコードクローン検出法 5

2.2 コードクローン検出ツール CCFinder 7

2.2.1 概要 7

2.2.2 コードクローン検出処理手順 9

2.2.3 検出例 9

2.3 コードクローン分析環境 Gemini 11

2.3.1 コードクローン検出部 12

2.3.2 クローンペア管理部 12

2.3.3 メトリクス管理部 12

2.3.4 ソースコード管理部 12

2.3.5 サブシステム間の提携 13

2.4 これまでの経験から言えること 13

3 提案手法 **14**

3.1 メトリクスの説明 14

3.2 提案手法の説明 15

3.2.1 コードクローンの分布に着目した分類方法 15

3.2.2 コードクローンの特徴を表す指標 16

4 適用実験 **17**

4.1 各カテゴリの大まかな傾向を調べるための実験 18

4.2 カテゴリの汎用性を評価する実験 19

4.3 各カテゴリの特徴を評価する実験 22

5 まとめと今後の課題 **32**

参考文献 **34**

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，ソフトウェアの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」による安易な機能的再利用の際に発生する．例えば，コード片にバグが含まれていた場合，そのコード片の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しなければならない．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，全てのコードクローンを検出することが必要である．

これまでに様々なコードクローン検出法が提案されている．我々の研究グループでもコードクローン検出ツール CCFinder[15] と分析環境 Gemini[21][22] 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 [23].

しかし，CCFinder が検出するのは，ユーザが必要とするコードクローンだけではない．それは，調査の関心がないコードクローン，例えば定型処理のような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，コードクローン分析環境 Gemini 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，ファイル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の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カテゴリに分類されたコードクローンは，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．

分類ごとの特徴を用いることで，ユーザは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になり，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，コードクローンに対する諸定義，コードクローン検出ツール CCFinder，コードクローン分析環境 Gemini について説明する．3 節では，コードクローンに関するメトリクスの説明と，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　準備

2.1　コードクローンと既存のコードクローン検出法

コードクローンとは，ソースコード中に含まれる同一もしくは類似したコードのことであり，いわゆる“ 重複したコード ”のことである．

コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものがある [8][15].

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，ゼロからコードを書くよりも既存コードをコピーして部分的な変更を加える方が信頼性が高いということもあり，実際には，コピーとペーストによる場当たり的な既存コードの再利用が多く存在する．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理，例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

偶然

単純に偶然一致してしまう場合もあるが，大きなコードクローンになる可能性は低い．

もしコードクローンが存在した場合には，一般的にコードの変更等が困難であるといわれ，保守容易性の低下の一因となっている．このようなコードクローンによる問題に対処する方法としては，

- コードクローン情報の文書化を行うこと変更の一貫性を保つ
- コードクローンを自動で検出する

の 2 つがある [23]．しかし，コードクローン情報の文書化には全てのコードクローンに対する情報を常に最新に保つことに非常に手間がかかるため，現実的に困難である．そこで，これまでにさまざまなコードクローン検出手法やツールが提案されている [1][2][3][4][5][6][7][8][10][15][17][18][19][20]．それぞれの手法やツールの特徴は次のようになっている (我々の開発した CCFinder は次節 2.2 参照)．

Covet

文献 [19] で定義された種々の特徴メトリクスの幾つかのメトリクス値を比較することによって，コードクローン検出を行う．検出対象言語は Java である．

CloneDR[8]

抽象構文木 (AST) の節点を比較することによってコードクローン (類似部分木) の検出を行う．また，部分的に異なっているコードクローンも検出することが可能であり，自動的に等価なサブルーチンやマクロに置き換えることも可能である．検出対象言語は，C，C++，COBOL，Java，Progress である．

Dup[2][3][4]

ユーザ定義名のパラメータ化を行った後，行単位の比較によりコードクローンを検出する，マッチングアルゴリズムには，サフィックス木探索 [11] を用いているため線形時間で解析可能である．

Duploc[10]

前処理として，空白やコメント等を取り除いた後，行単位 (のハッシュ値) での表検索を用いた比較によってコードクローンを検出する．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコードの参照支援を行う．検出対象言語は，C，COBOL，Python，Smalltalk である．

JPlag[20]

ソースコードを字句解析し，トークン単位での比較を行う．プログラム盗用の検出を目的として開発され，プログラム間の類似率を検出する．検出対象言語は，C，C++，Java である．

Komondoor らの手法 [17]

関数等にまとめるのに適したコードクローンを抽出を目的として，プログラム依存グラフ (PDG) 上での各節点の比較を行うことでコードクローン (同型 (isomorphic) 部分グラフ) を検出する．文字列比較や抽象構文木等を用いた検出法では発見できなかった非連続コードクローンや，対応行の番号が異なるクローン，互いに絡み合ったクローン等を検出可能である．[17] で作成されたツールの検出対象言語は，C である．

Krinke の手法 [18]

AST や Traditional PDG に似た Fine-grained PDG というグラフ上での類似 (similar) 部分グラフ (同型部分グラフではない) を検出することで，コードクローンが存在すると思しき場所を検出する．試作ツールの検出対象言語は，C である．

SMC[5][6][7]

まず特徴メトリクスによってソースコードをコードクローンと思しきメソッドに絞り込む．次に絞り込まれたメソッドのペアに対し，表検索を用いることでメソッド単位のコードクローンを検出する．特徴メトリクスによって絞り込まれているため，実用上ほぼ線形時間で解析可能である．また検出されたペアのメソッドは，特徴により 18 種類に分類される．さらにそれぞれの分類については共通メソッドへの書き換え指針が示されている．

MOSS[1]

検出アルゴリズムは公開されていない．JPlag 同様，プログラム盗用の検出を目的として開発された．検出対象言語は，Ada，C，C++，Java，Lisp，ML，Pascal，Scheme である．

いずれの手法，ツールにおいても提案者によってコードクローンの定義が微妙に異なっており，検出されるコードクローンが異なっている．つまり，コードクローンの定義とは検出アルゴリズムそのものによって定義される．Burd ら [9] も，CloneDR，Covet，JPlag，Moss，そして我々の開発した CCFinder を含めた 5 つのツールを用いて，それぞれ検出されるコードクローンの比較を行っているが，すべての面において他のツールより優れているツールはなく，使う場面に応じて，適切なツールを選ぶことが必要となってくると述べている．

2.2 コードクローン検出ツール CCFinder

2.2.1 概要

あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α は β 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，(α，β) を**クローンペア**

(図 1 参照) と呼ぶ．α，β 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 α，β 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．

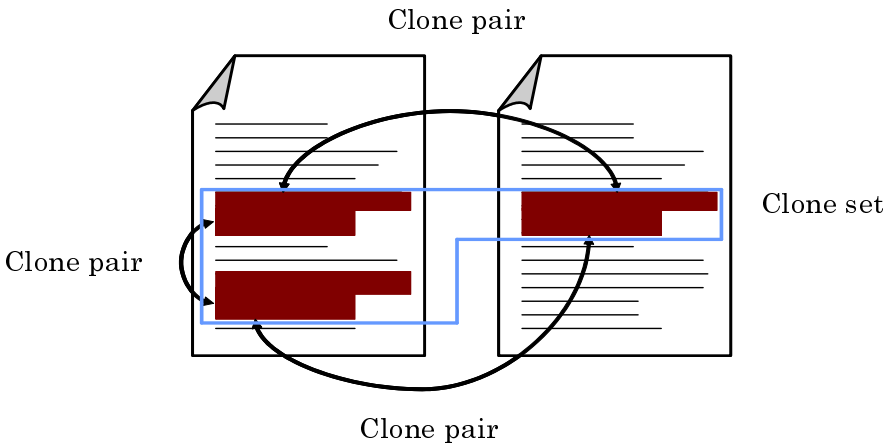


図 1: クローンペアとクローンセット

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 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である [23].

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．

2.2.2 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．

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．

2.2.3 検出例

実際に，CCFinder がどのようなコードクローンを検出するのか例を示す．図 2 は説明のための Java ソースコードである．このソースコードには，互いに似通った 2 つのメソッドが含まれ，左端には行番号が付されている．ここで，最小一致トークン数を 5 トークンに定め，図 2 のソースコードに対しコードクローン検出を行うと，図 2 中の A1(4 行目-6 行目)

と A2(16 行目-17 行目)，B1(8 行目-10 行目) と B2(20 行目-22 行目)，そして C1(12 行目) と C2(25 行目) がそれぞれクローンペアとして検出される．それぞれのクローンペアの長さは順に 7,18,6 トークンとなっている．見ての通り，A1 と A2 の間，B1 と B2 の間には次のような幾らかの違いが含まれているがコードクローンとして検出可能となっている．

- 名前空間の違い (e.g.“ org.apache.regexp.RE ”と“ RE ”).
- 変数名の違い (e.g.“ pat ”と“ exp ”).
- 改行とインデントの違い
- 中括弧表記の違い

これらの違いは，2.2.1 節で述べた目的のため，CCFinder のトークン変換処理によって吸収されている．

```
     1.   static void foo() throws RESyntaxException
     2.   {
     3.     String a[] = new String [] {“123,400”,”abc”};
A1   4.     org.apache.regexp.RE pat =
A1   5.               new org.apache.regexp.RE(“[0-9,]+”);
A1   6.     int sum = 0;
     7.     for (int i = 0; i < a.length; i++)
B1   8.     {
B1   9.       if (pat.match(a[i])){
B1   10.          sum += Sample.parseNumber(pat.getParen(0));}
     11.    }
C1   12.    System.out.println(“sum = ” + sum);
     13.  }
     14.  static void goo(String [] a) throws RESyntaxException
     15.  {
A2   16.    RE  exp  = new RE(“[0-9,]+”);
A2   17.    int  sum = 0;
     18.    int  i      = 0;
     19.    while (i < a.length)
B2   20.    {
B2   21.      if (exp.matcn(a[i]))
B2   22.          sum += parseNumber(exp.getParen(0));
     23.      i++;
     24.    }
C2   25.    System.out.println(“sum = ” + sum);
     26.  }
            :
            :
```

図 2: コードクローン検出例

2.3 コードクローン分析環境 Gemini

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．システムの構成を図 3 に示す．Gemini は主に 5 つのコンポーネント:

1. コードクローン検出部 (Code clone detecter),
2. クローンペア管理部 (Clone pair manager),
3. メトリクス管理部 (Metrics manager),
4. ソースコード管理部 (Source code manager),
5. ユーザインターフェース,

で構成されている．まず始めに，コードクローン検出部にソースファイルが入力され，コードクローンを検出する．次に，クローンペア管理部と，メトリクス管理部がその解析結果であるコードクローン情報を各インターフェースを通して視覚化する．それらのインターフェース上では，ユーザは任意のクローンペア (あるいは，クローンセット) を選択することができ，その選択によって，実際のソースコードをソースコード管理部とそのインターフェースを通して参照することができる．

次節から図 3 における各コンポーネントについて順に述べる．

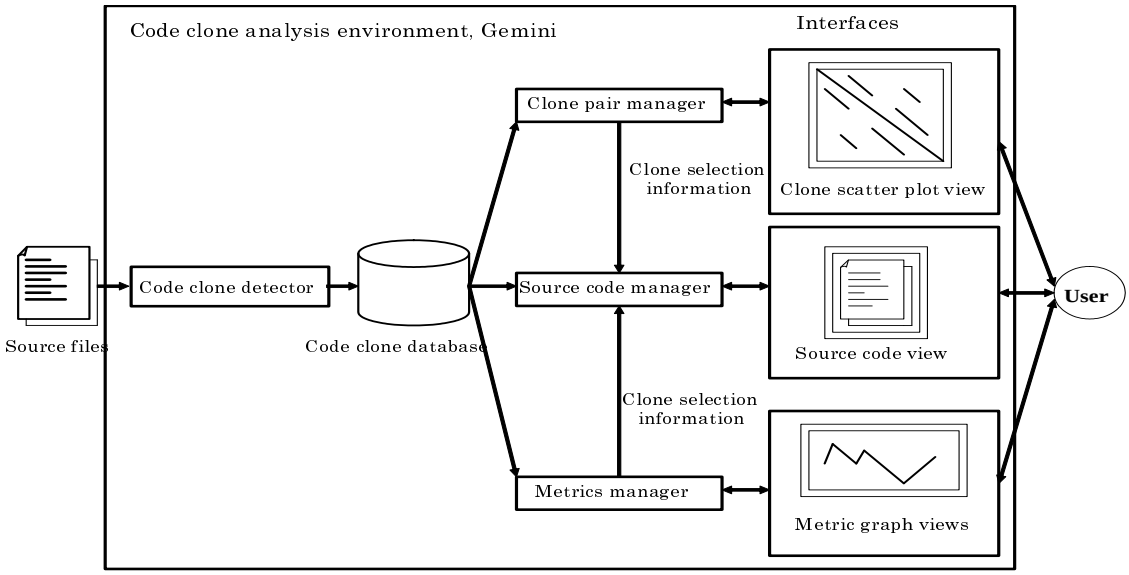


図 3: Gemini の構成

2.3.1 コードクローン検出部

解析対象のファイルや，コードクローン検出のパラメータを管理する．CCFinder を利用して検出したコードクローンの位置情報も管理する．

2.3.2 クローンペア管理部

クローンペア位置情報を基にして，利用者の要求に応じて，クローン散布図を表示する．クローン散布図上での GUI による操作として，拡大・縮小，および，任意のクローンペア集合を「選択状態」にすること，がある (選択状態，後述する他のサブシステムと連携した操作で使われる)．クローン散布図はソースコードのどの部分にクローンペアが存在するのかを示す図である．一目でソースコード中のコードクローンの分布状況がわかるので，コードクローン解析の初期段階では非常に有効な解析手段となりうる．

クローン散布図

Gemini が表示するクローン散布図の簡単なモデルを図 4 に示す．散布図の原点は左上隅にあり，水平軸，垂直軸は，それぞれソースファイルの並び (f1 から f6) に対応している．両軸上で，原点から順に，それぞれのソースファイルに含まれるトークンが並んでいる．座標平面内に点がプロットされている部分は，その両軸の対応するトークンが一致することを意味する．したがって，散布図の主対角線は，両軸同じ位置のトークンを比較することになり，すべての点がプロットされることになる．また，点の分布は主対角線に対して線対称となる．一定の長さ (CCFinder で設定される最小一致トークン数) 以上の対角線分が，検出されたクローンペアである．図 4 では，f1 から f6 までのファイルが，それぞれ 3 つのトークンを含んでいる．ファイルの並びはファイル名の辞書順となっている．検出されるクローンペアは f1 と f6 に含まれる“ ab ”というトークン列と，f3 と f6 に含まれる“ bc ”というトークン列である．

2.3.3 メトリクス管理部

クローンペアの位置情報から，6 種のメトリクスを算出する．利用者の要求に応じて，メトリクスグラフによってメトリクス計測値を表示する．メトリクスグラフは多次元平行座標表現 [16] を用いている．Gemini で用いているメトリクスの説明については，次章 3.1 で述べる．

2.3.4 ソースコード管理部

指定されたソースファイルの内容をソースコードビューを通じて表示する．

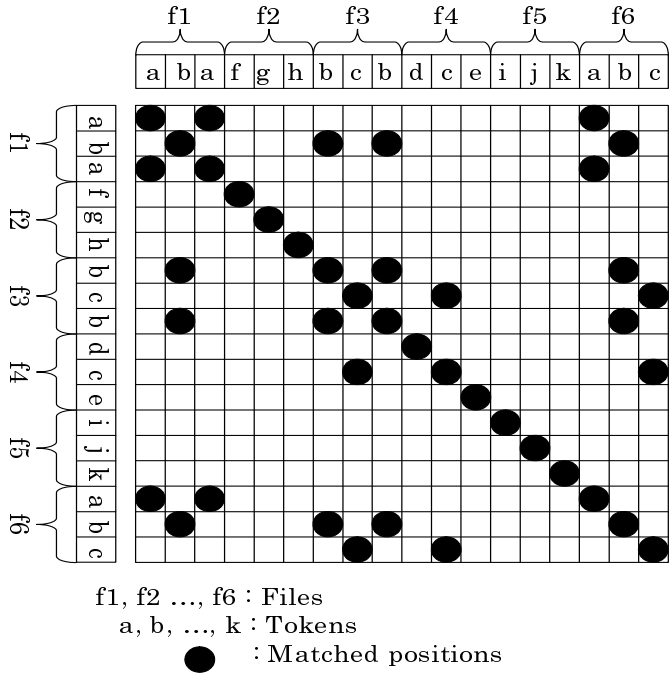


図 4: クローン散布図モデル

2.3.5　サブシステム間の提携

クローンペア管理部，ソースコード管理部，メトリクス管理部の間で「選択状態」を用いた提携が可能になっている．この提携により，利用者は例えば，(1) クローン散布図で目立つ部分のクローンペアを選択した後，そのソースコードを表示する，(2) メトリクスグラフで要素数の多いクローンセットを選択した後，そのソースコードを表示する，(3) メトリクスグラフで長いクローンセットだけを選択した後，クローン散布図ではそれらがどこにあるのか調べる，といった対話的操作を行うことができる．

2.4　これまでの経験から言えること

リファクタリングを行う場合や，設計情報との一貫性を確認する場合にコードクローン情報が用いられている．しかし，大規模なプログラムになると，コードクローンの数は膨大になってしまう．そのため，利用者が必要な情報を見つけることが非常に困難である．もしコードクローンを特徴に応じて分類できるならば，利用者は必要な情報を容易に得ることができる．

次節 3 節ではコードクローンの分布情報を用いることでその特徴を抽出する手法を提案する．

3　提案手法

本研究では，コードクローンを特徴に応じてカテゴリに分類する手法を提案する．コードクローンの特徴を定量的に表すために，Gemini で用いられているメトリクスを利用する．

3.1　メトリクスの説明

Gemini で用いられているメトリクスは，RAD(S)，LEN(S)，RNR(S)，NIF(S)，POP(S)，DFL(S) の 6 つである．以下に各メトリクスについての説明を行う．

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 5 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．

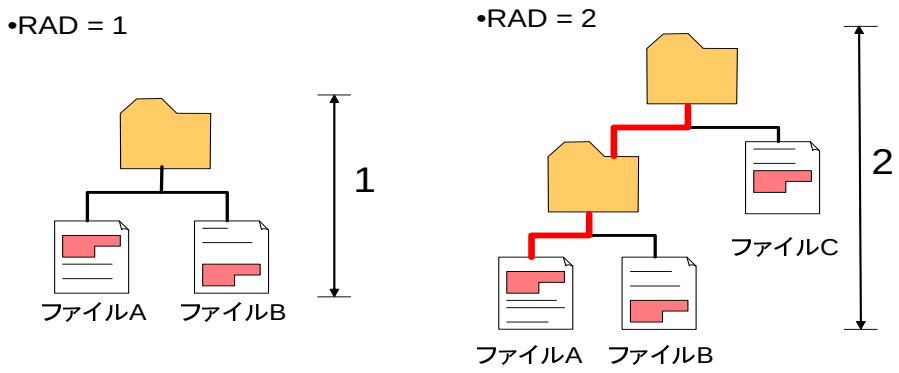


図 5: RAD の値の例

LEN(S)

クローンセット S に含まれるコードクローンの平均トークン数を表す．この値が大きいほど，コードクローンのサイズが大きいことになる．

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．

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

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．

POP(S)

クローンセット S に含まれるコードクローンの数を表す．これは NIF とは違い，同じファイル中にクローンセット S に含まれるコードクローンが複数存在する場合には，その数の分カウントする．

DFL(S)

クローンセット S に含まれるコードクローンをサブルーチンなどに集約した場合に，削減されるトークン数の予測値を表す．この値が大きいほど，プログラムサイズ面での集約の効果が大きいことを表す．

3.2 提案手法の説明

本手法では，コードクローンを 3.1 節で述べたメトリクスを用いて複数のカテゴリに分類する．また，各カテゴリを評価するために用いた指標について説明する．

3.2.1 コードクローンの分布に着目した分類方法

3.1 節で述べたメトリクスのうち，RAD，NIF はクローンの分布の特徴を表すメトリクスである．本手法ではこの 2 つのメトリクスの組み合わせによりカテゴリ分けを行う．その分け方のモデルを図 6 に表す．それぞれのカテゴリの特徴は次のようになる．

- local (RAD 値低，NIF 値低): 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): 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): コードクローンがディレクトリ階層上遠くの少数のファイルに存在する．

- global (RAD 値高，NIF 値高): コードクローンがプログラム広範囲の多数のファイルに存在する．

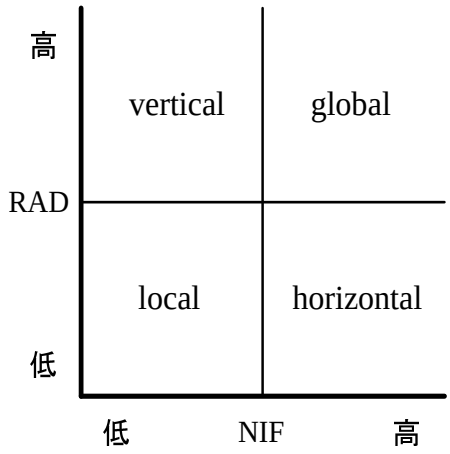


図 6: RAD 値，NIF 値によるカテゴリ分けモデル

3.2.2 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，分類したカテゴリの特徴を評価する．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 7 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．

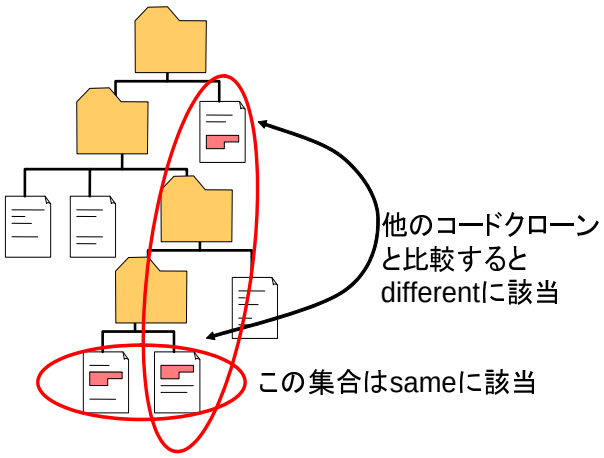


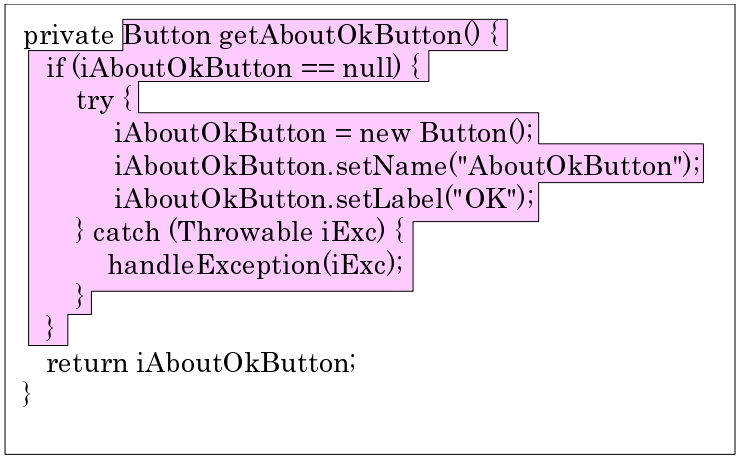
図 7: 複数の指標に該当する例

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
Webmail	Java	154	111	約 18,000 行
httpunit	Java	1,231	193	約 43,000 行
Art of Illusion	Java	1,856	409	約 120,000 行
CppUnit	C++	86	159	約 20,000 行
SWIG	C++	2,158	208	約 85,000 行
Small Device C Compiler	C	7,550	871	約 367,000 行
Sketch	C	196	43	約 22,000 行

表 1: 調査対象データサイズ

4 適用実験

本節では，前章で述べた提案手法を定義する過程で行った実験について説明する．実行環境は，CPU は Pentium4 2.8GHz，メモリは 1GB，OS は Windows XP である．本実験では，CCFinder の検出する最小のコードクローンの大きさは 30 トークンとした．30 トークンとは，我々がこれまでに CCFinder を用いて行ってきた経験から導かれた値である．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．



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

図 8: local に属するコードクローンの例

4.1 各カテゴリの大まかな傾向を調べるための実験

まず始めに各カテゴリ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[12] 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．カテゴリ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．カテゴリごとの結果は以下のようになった．

- local: RNR の値が低いコードクローンは，ロジックが単純な連続した命令であった．RNR の値が高いコードクローンは，あるデータ構造の要素に対して処理を行うメソッド内に見られた．例えば，コードクローンが GUI 部品のメソッド内に存在していた．図 8 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．

- 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実装しているファイル中に見られた．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メソッド内に見られた．

- vertical: RNR の値が低いコードクローンは，カテゴリ 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

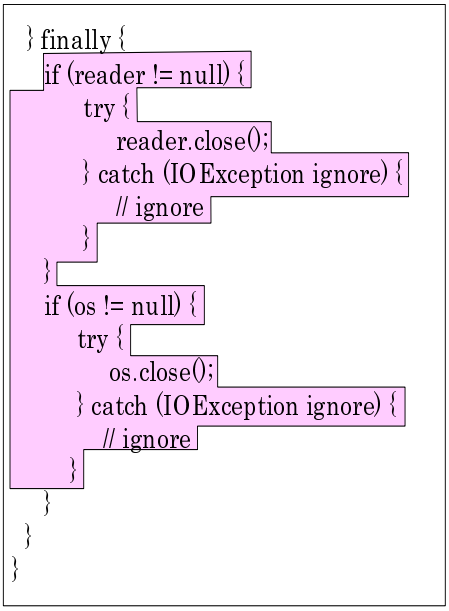


図 9: global に属するコードクローンの例

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 9 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．

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

上記の調査結果から，各カテゴリには以下の特徴があると言える．

- local: コードクローンが同じデータ構造を用いた処理に存在する．
- horizontal: コードクローンが 1 つのパッケージ内で親クラス下の多数の子クラスが実装している同様の処理に存在する．
- vertical: コードクローンが類似した構造のパッケージ内に存在する．
- global: コードクローンが一般的な定型処理を構成している．

4.2 カテゴリの汎用性を評価する実験

次に，前節で述べた各カテゴリの特徴の汎用性を調べるために，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 書かれた他のソフトウェアを調査し，ま

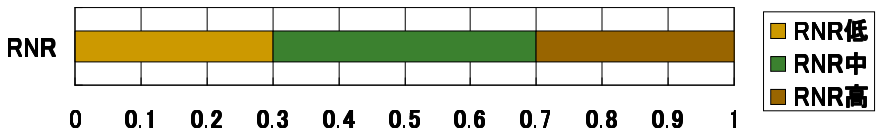


図 10: RNR の区分

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前節の Ant に加え，オープンソース・ソフトウェア開発サイト SourceForge.net[13] から入手した．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．

RAD，NIF の高低の閾値を RAD，NIF のそれぞれの最大値の半分とした．また，それぞれのカテゴリにおいて RNR の値によって区分することにした．これは，前節での RNR の値が低い場合に各カテゴリ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 10 に示す．

また，検出されたクローンセット数が多いソフトウェアは，POP の値を一定値以上のクローンセットについて調査するようにフィルタリングした．規模がある程度大きいコードクローンについて調べることで，コードクローンの特徴を調査しやすくするためである．

調査を行った結果，Java と C，C++のプログラムにおけるカテゴリ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．以下にそのカテゴリごとの結果を述べる．

- local: コードクローンが，あるデータ構造の要素に対して処理を行う関数内に存在した．また，コードクローンが同一関数内で意図的に繰り返されている場合も見られた．
- horizontal: コードクローンが，1 つのパッケージ内で親クラス下の多数の子クラスが実装している同様の処理の中に存在した．
- vertical: コードクローンが，類似した構造のパッケージ下に存在した．
- global: コードクローンが，プログラミング上よく見られる定型処理であった．例として図 11 を挙げる．

また，RNR の値が低い場合に各カテゴリで共通して見られた特徴，つまりロジックが単純な連続した命令が，クローンセット中に RNR の値の高低によってどの程度見られたかを表 2，表 3 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．

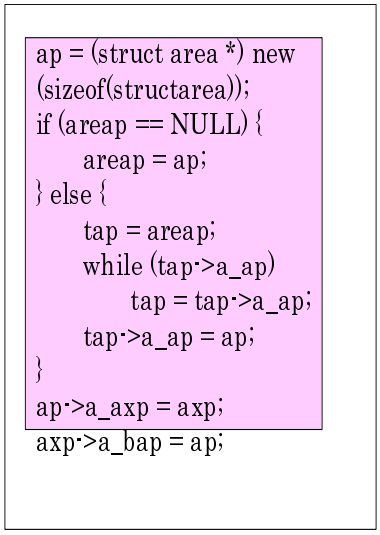


図 11: 定型処理の例

ソフトウェア名		Ant	Webmail	httpunit	Art of Illusion
フィルタ		POP3 以上	なし	POP4 以上	POP3 以上
local (RAD 低,NIF 低)	RNR 低	94(94)	66(66)	6(6)	130(98)
	RNR 中	131(80)	10(8)	3(1)	172(98)
	RNR 高	305(24)	69(12)	26(4)	300(23)
horizontal (RAD 低,NIF 高)	RNR 低	1(1)	0	0	0
	RNR 中	1(1)	0	0	0
	RNR 高	9(0)	4(2)	1(0)	0
vertical (RAD 高,NIF 低)	RNR 低	17(17)	0	2(2)	34(34)
	RNR 中	12(3)	0	0	9(9)
	RNR 高	4(0)	3(0)	1(0)	11(1)
global (RAD 高,NIF 高)	RNR 低	3(3)	0	7(7)	6(6)
	RNR 中	0	0	0	1(1)
	RNR 高	4(0)	2(1)	0	0

表 2: Java のソフトウェアにおけるロジックが単純な連続した命令の分布

4.3 各カテゴリの特徴を評価する実験

次に，カテゴリごとの特徴を評価するために，3.2 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．4.1.2 節で実験を行ったソフトウェアに対して，カテゴリごとの特徴を指標を用いて評価する実験を行った．RAD，NIF の高低，RNR の閾値は，4.1.2 節の実験と同じ条件とした．結果は表 4 から表 11 となった．

表の値は，カテゴリごとに 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．

各カテゴリ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複数の子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．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 に該当するコードクローンが生成される可能性がある．

- global: このカテゴリに属するクローンセットは基本的に different に該当している．定型処理というの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．

ソフトウェア名		CppUnit	SWIG	Small Device C Compiler	Sketch
フィルタ		なし	POP3 以上	POP10 以上	なし
local (RAD 低,NIF 低)	RNR 低	24(24)	237(228)	162(152)	41(40)
	RNR 中	19(16)	230(142)	109(94)	17(9)
	RNR 高	32(3)	298(69)	68(38)	128(0)
horizontal (RAD 低,NIF 高)	RNR 低	1(0)	20(20)	64(64)	0
	RNR 中	3(2)	8(4)	0	0
	RNR 高	3(0)	6(0)	0	0
vertical (RAD 高,NIF 低)	RNR 低	0	14(14)	27(19)	6(5)
	RNR 中	0	13(13)	5(5)	0
	RNR 高	0	5(1)	0	2(0)
global (RAD 高,NIF 高)	RNR 低	1(0)	14(14)	6(6)	0
	RNR 中	0	0	1(1)	0
	RNR 高	3(0)	0	0	2(0)

表 3: C，C++のソフトウェアにおけるロジックが単純な連続した命令の分布

		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: Ant の評価結果

		Webmail			
フィルタ					
特徴		all	same	similar	different
local	RNR 低	66	0	4(6.1 %)	0
(RAD 低,NIF 低)	RNR 中	10	3(30.0 %)	1(10.0 %)	0
合計 145	RNR 高	69	27(39.1 %)	28(40.6 %)	6(8.7 %)
horizontal	RNR 低	0	0	0	0
(RAD 低,NIF 高)	RNR 中	0	0	0	0
合計 4	RNR 高	4	1(25.0 %)	3(75.0 %)	0
vertical	RNR 低	0	0	0	0
(RAD 高,NIF 低)	RNR 中	0	0	0	0
合計 3	RNR 高	3	3(100.0 %)	0	0
global	RNR 低	0	0	0	0
(RAD 高,NIF 高)	RNR 中	0	0	0	0
合計 2	RNR 高	2	0	2(100.0 %)	0

表 5: Webmail の評価結果

		httpunit			
フィルタ		POP4 以上			
特徴		all	same	similar	different
local	RNR 低	6	3(50.0 %)	3(50.0 %)	0
(RAD 低,NIF 低)	RNR 中	3	1(33.3 %)	2(66.7 %)	0
合計 35	RNR 高	26	1(3.8 %)	25(96.2 %)	0
horizontal	RNR 低	0	0	0	0
(RAD 低,NIF 高)	RNR 中	0	0	0	0
合計 1	RNR 高	1	1(100.0 %)	0	0
vertical	RNR 低	2	0	0	2(100.0 %)
(RAD 高,NIF 低)	RNR 中	0	0	0	0
合計 3	RNR 高	1	1(100.0 %)	0	0
global	RNR 低	7	0	0	7(100.0 %)
(RAD 高,NIF 高)	RNR 中	0	0	0	0
合計 7	RNR 高	0	0	0	0

表 6: httpunit の評価結果

		Art of Illusion			
フィルタ		POP3 以上			
特徴		all	same	similar	different
local	RNR 低	130	56(43.1 %)	81(62.3 %)	20(15.4 %)
(RAD 低,NIF 低)	RNR 中	172	121(70.3 %)	71(41.3 %)	3(1.7 %)
合計 602	RNR 高	300	238(79.3 %)	109(36.3 %)	2(0.7 %)
horizontal	RNR 低	0	0	0	0
(RAD 低,NIF 高)	RNR 中	0	0	0	0
合計 0	RNR 高	0	0	0	0
vertical	RNR 低	34	1(2.9 %)	19(55.9 %)	30(88.2 %)
(RAD 高,NIF 低)	RNR 中	9	2(22.2 %)	5(55.6 %)	4(44.4 %)
合計 54	RNR 高	11	7(63.6 %)	3(27.3 %)	8(72.7 %)
global	RNR 低	6	4(66.7 %)	5(83.3 %)	6(100.0 %)
(RAD 高,NIF 高)	RNR 中	1	0	1(100.0 %)	1(100.0 %)
合計 7	RNR 高	0	0	0	0

表 7: Art of Illusion の評価結果

		CppUnit			
フィルタ					
特徴		all	same	similar	different
local	RNR 低	24	0	8(33.3 %)	0
(RAD 低,NIF 低)	RNR 中	19	1(5.3 %)	12(63.2 %)	0
合計 75	RNR 高	32	1(3.1 %)	29(90.6 %)	2(6.3 %)
horizontal	RNR 低	1	0	1(100.0 %)	0
(RAD 低,NIF 高)	RNR 中	3	1(33.3 %)	0	0
合計 7	RNR 高	3	3(100.0 %)	0	0
vertical	RNR 低	0	0	0	0
(RAD 高,NIF 低)	RNR 中	0	0	0	0
合計 0	RNR 高	0	0	0	0
global	RNR 低	1	0	0	1(100.0 %)
(RAD 高,NIF 高)	RNR 中	0	0	0	0
合計 4	RNR 高	3	3(100.0 %)	0	0

表 8: CppUnit の評価結果

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

表 9: SWIG の評価結果

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

表 10: Small Device C Compilar の評価結果

		Sketch			
フィルタ					
特徴		all	same	similar	different
local	RNR 低	41	1(2.4 %)	2(4.9 %)	18(43.9 %)
(RAD 低,NIF 低)	RNR 中	17	1(5.9 %)	8(47.1 %)	3(17.6 %)
合計 186	RNR 高	128	18(14.1 %)	96(75.0 %)	14(10.9 %)
horizontal	RNR 低	0	0	0	0
(RAD 低,NIF 高)	RNR 中	0	0	0	0
合計 0	RNR 高	0	0	0	0
vertical	RNR 低	6	0	1(16.7 %)	5(83.3 %)
(RAD 高,NIF 低)	RNR 中	0	0	0	0
合計 8	RNR 高	2	0	1(50.0 %)	2(100.0 %)
global	RNR 低	0	0	0	0
(RAD 高,NIF 高)	RNR 中	0	0	0	0
合計 2	RNR 高	2	0	2(100.0 %)	0

表 11: Sketch の評価結果

異なるソフトウェアでも，カテゴリ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，各カテゴリ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，カテゴリ 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，カテゴリ 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う．

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，分類したカテゴリ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，分類したカテゴリの特徴を評価した．その評価の結果，本手法によって同じカテゴリに分類されたコードクローンは，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したカテゴリ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，分類したカテゴリ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．以上より，今後の課題としては，分類の詳細化，ツールとしての実装，実際の開発現場での適用実験と評価などが挙げられる．

謝辞

本研究を通して，常に適切な御指導，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致します.
本研究を通して，常に適切な御指導，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 楠本 真二 教授に心から感謝致します.
本研究を通して，常に適切な御指導，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 助教授に心から感謝致します.
本研究を通して，逐次適切な御指導，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 肥後 芳樹 氏に深く感謝致します.
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致します.

参考文献

[1] A. Aiken, “ A System for Detecting Software Plagiarism (Moss Hompage) ”, `http://www.cs.berkley.edu/~aiken/moss.html` `[Last visited 1st Feb. 2003]`

[2] B. S. Baker, “ A Program for Identifying Duplicated Code ”, *Computing Science and Statistics*, 24:pp.49-57, 1992.

[3] B. S. Baker, “ On Finding Duplication and Near-Duplication in Large Software Systems ”, *Proceedings of the 2nd Working Conference on Reverse Engineering*, pp.86-95, 1995.

[4] B. S. Baker, “ Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance ”, *SIAM Journal on Computing*, 26(5):pp.1343-1362, 1997.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontoginannis, “ Advanced Clone-Analysis to Support Object-Oriented System Refactoring ”, *Proceedings of the 7th Working Conference on Reverse Engineering*, pp.98-107, 2000.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontoginannis, “ Measuring Clone Based Reengineering Opportunities ”, *Proceedings of the 6th IEEE International Symposium on Software Metrics*, pp.292-303, 1999.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontoginannis, “ Partial Redesign of Java Software Systems Based on Clone Analysis ”, *Proceedings of the 6th IEEE International Working Conference on Reverse Engineering*, pp.326-336, 1999.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “ Clone Detection Using Abstract Syntax Trees ”, *Proceedings of the 14th IEEE International Conference on Software Maintenace-1998*, pp.368-377, 1998.

[9] E.Burd, and J. Bailey, “ Evaluating Clone Detection Tools for Use during Preventative Maintenance ”, *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp.36-43, 2002.

[10] S. Ducasse, M. Rieger, and S. Demeyer, “ A Language Independent Approach for Detecting Duplicated Code ”, *Proceedings of the 15th IEEE International Conference on Software Maintenance-1999*, pp.109-118, 1999.

[11] D. Gusfield, *Algorithms on Strings, Trees, And Sequence*, Cambridge University Press, 1997.

[12] Ant, **http://ant.apache.org/**, [Last visited 17 Feb. 2006].

[13] SorceForge.net, **http://sourceforge.net/**, [Last visited 17 Feb. 2006].

[14] 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):pp.47-54, 2001.

[15] 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):pp.654-670, 2002.

[16] 加藤 博己, “ データベースのビジュアルな検索と分析 (OLAP) ”, IPSJ Magazine Vol.41 No.4 pp.363-368, 2000.

[17] R. Komondoor, and S. Hirwitz, “ Using Slicing to Identify Duplication in Source Code ”, *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[18] J. Krinke, “ Identifying Similar Code with Program Dependece Graphs ”, *Proceedings of the 8th Working Conference on Reverse Enngineering*, pp.562-584, 2001.

[19] J. Mayland, C. Leblanc, and E. Merlo, “ Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics ”, *Proceedings of the 12th IEEE International Conference on Software Maintenace-1996*, pp.244-253, 1996.

[20] L. Prechelt, G. Malpohl, and M. Philippsen, “ Finding plagiarisms among a set of programs with JPlag ”, *resubmitted to Journal of Universal Computer Science*, 2001.
`http://www.ipd.uka.de/~prechelt/Biblio/#jplag`
`[Last visited 1 Feb. 2002]`

[21] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “ Gemini: Maintenance Support Environment Based on Code Clone Analysis ”, *Proceedings of the 8th International Symposium on Software Metrics*, pp.67-76, 2002.

[22] 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, pp.863-871, 2003.

[23] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “ On Detection of Gapped Code Clones using Gap Locations ”, *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, pp.327-336, 2002.