コメント箇所推薦のための類似コード片検出方法の提案

M2020SE003　小山哲明
指導教員：蜂巣吉成

1　はじめに

プログラム中のコメントは，開発者がソースコードを理解するための有効な手段の一つである．コメントは適切な箇所に適切な内容で記述する必要がある．しかし，開発者のレベルやプロジェクト内のコーディング規約などによってコメントをつけるべき箇所を一概には決められない．本研究では次のコードをコメントをつける箇所としてあげる．

箇所 1 コメントがついているコードと類似したコード

箇所 2 類似したコード群の中で，一部異なる処理をしているコード

箇所 1 は，既にコメントがついているコードは開発者がそのコードの理解のためにコメントが必要として記述したとすると，同程度の難解さをもつ類似したコードにもコメントをつけて理解支援すべきであるという考えに基づく．箇所 2 については，異なる処理が間違って記述がされているのか意図して記述がされているのかをコメントをつけて記述の意図を明確にすべきであると考えた．

本研究は，上記 2 箇所を検出するために類似したコード片を検出する方法を提案する．本研究では制御構造や変数名などに着目してコード片を抽象化し類似度を計算する．抽象化には構文レベルと名前レベルの 2 つを用いて，構文レベルでは制御文や文，演算子を扱い，名前レベルでは識別子を単語レベルに区切ったものを扱う．構文レベルを用いることでコードクローンのような似ている処理の検出が可能であると考える．名前レベルを用いることで制御構造が異なるが難解さが同等なコード片の検出が可能であると考える．箇所 2 を検出するために段階的な抽象度が必要であるので，構文レベルでは 3 つの抽象度用いる．対象とするコメントはコード中に記述された行コメントで，コードのコメントアウト，javadoc や TODO コメントといったコメントは対象としない．

2　関連研究

類似処理検出の研究としてコードクローン検出の研究 [1][2][3][4] がある．コードクローンとは，既存のコードをコピーアンドペーストによって作られる，一致または類似したコード片である．Kamiya ら [1] は変換ルールとトークンベースを用いたコードクローン検出ツール CCFinder を提案した．関数名や変数名が異なる場合でも検出をすることが可能である．コードクローンは以下の 3 つのタイプに分類される．

Type1 空白文字や改行およびコメント文の差異を除き一致するコードクローン

Type2 Type1 に加え，識別子名やリテラルの差異を除き一致するコードクローン

Type3 Type2 に加え，式・文の追加や削除を含むが，類似するコードクローン

コードクローンでは Listing1 や Listing2 のような構文要素が異なる類似処理を検出することができないと考えられる．理由として，Listing1 の 5 行目と Listing2 の 4 行目の制御文が異なるからである．本研究ではコードクローンでは検出できない Listing1 や Listing2 の類似処理を検出する方法を提案する．

Marcus ら [2] は識別子とコメントを用いたコードクローン検出方法の提案を行なった．リストの抽象データ型のような高いレベルの抽象度で表現されるクローンをハイレベルコンセプトクローンとして検出する方法の提案を行なった．データや制御フローが異なる 2 つの異なる実装を判定することができないが，意味的な類似性が一致するクローンを検出した．またコメントが存在せず，識別子の名前が異なる場合や構造や機能が類似していない場合には検出をすることができない．

3　コメント分析

コードの理解支援のためのコメントについて分析する．箇所 1 は Linsting1 と Linsting2 のようなコメントがあるコード片に対して，構文や処理の内容が同じ処理のコメントがないコード片である．Listing1 と Listing2 は kilo.c[6] に記述されている行を挿入する処理として類似した処理である．

Listing 1　類似処理においてコメントがある例

```
1 /* If the row where the cursor is currently located
       does not exist in our
2   * logical representaion of the file, add enough empty
       rows as needed. */
3 if (!row) {
4   while(E.numrows <= filerow)
5   editorInsertRow(E.numrows,"",0);
6 }
```

Listing 2　類似処理においてコメントがない例

```
1 if (!row) {
2   if (filerow == E.numrows) {
3     editorInsertRow(filerow,"",0);
4     goto fixcursor;
5   }
6   return;
7 }
```

Listing1 は文字を挿入する関数 editorInsertChar 内部のコード片である．行が存在しない場合に必要に応じて行を追加を行うコード片である．Listing2 は関数 editorInsertNewline の内部のコード片である．行が存在しない場合に新しい新しい行を追加を行うコード片である．コードでは if と while という違いがあるが，if(!row) の本体で filerow と E.numrows を比較し，edirInsertRow 関数を呼び出しているという点で類似していると言える．あるプロジェクトで理解が難しい箇所にコメントが付けれられているとするならば，同程度の難解さを持つコードにも

同様にコメントをつけるべきである．箇所 2 は Listing3 と Listing4 のようなコード片である．

Listing 3 例外的な処理を行う例

```
1 if (*p == '\\') {
2     row->hl[i+1] = HL_STRING;
3     p += 2; i += 2;
4     prev_sep = 0;
5     continue;
6 }
```

Listing 4 類似処理の例

```
1 if (*p == '"' || *p == '\'') {
2     in_string = *p;
3     row->hl[i] = HL_STRING;
4     p++; i++;
5     prev_sep = 0;
6     continue;
7 }
```

2 つのコード片は kilo.c[6] に記述されているエディタのハイライトの処理を行うコード片として類似処理であり，*p に関する if 文の本体で，row->h の配列を添え字 i を使って，要素に HL_STRING を代入し，p, i を増やした後，prev_sep に 0 を代入し，continue している．一般に変数の値を 1 増やす処理はよく行われるが，Listing3 のように，2 増やす処理はそれほど多くない．Listing3 は h[i+1] で配列の要素を指定していることも注意が必要な処理と言える．kilo.c には Listing3 と同じ制御構造を持ち，変数の値に 2 増やす処理が合計 2 つあり，Listing4 と同じ制御構造を持ち，変数の値を 1 増やす処理が合計 3 つあった．

本研究ではコメント箇所推薦のための類似したコード片として Listing1 と Listing2 のような類似処理の検出と Listing3 と Listing4 のような類似処理中の例外的な処理の検出する方法を提案する．

4 コード理解支援コメントのための類似処理の検出の提案

4.1 類似するコード片の検出法の提案

コメントは制御文に対して書くことが多いと考え，本研究の対象とするコード片は関数内部の制御文とする．本研究で提案する手法はコード片に対して特徴となる構文要素を抽出し他のコード片と類似度を計算する方法である．図 1 は提案する手法の流れである．ソースコード群から TEBA[5] を用いて構文解析を行い制御文単位のコード片を扱う．それぞれのコード片に対して構文レベルと名前レベルの 2 つのレベルで抽象化を行い，総当たりで類似度を計算する．類似度計算には 2 つのレベルで抽象化したコード片に対してそれぞれの類似度計算の手法を用いる．構文レベルで抽象化したコード片に対しては要素の並びを考慮した編集距離であるレーベンシュタイン距離を用いた類似度計算を行う．名前レベルで識別子を抜き出したコード片に対しては並び順は関係ないと考え Jaccard 係数を用いた類似度計算を行う．また本研究ではコードとコメントが対応しているものとして扱う．

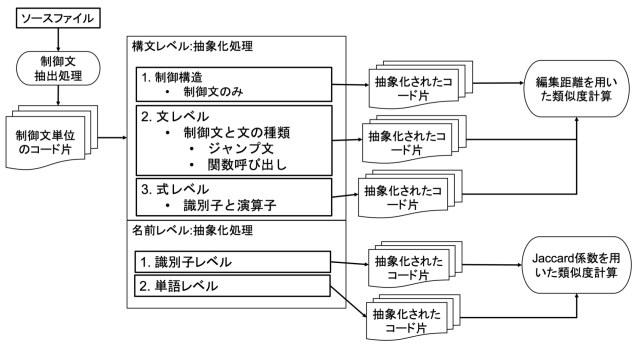

図 1 提案する手法の全体の流れ

4.2 コード片の抽象化

類似度を計算するための構成要素として構文レベルと名前レベルの 2 つの基準を持つ．2 つを基準を用いる理由は，構文レベルでは制御文や文の種類，演算子が同じであることは似た処理をしていることが多いと考え，名前レベルでは制御構造が異なるが類似した処理を検出することや制御構造が同じでかつ名前が同じである処理に対してより類似していると判定するためである．例えば Listing1 では 5 行目は while 文になっているが Listing2 では if 文とで異なっているので制御文によって類似度を判定することは難しい．そこで，if 文の条件式を取り上げ同じ変数が使われていることや関数 editorInsertRow の引数が同じであることから条件式や引数名を類似度検出のための構成要素として扱うことで類似したコード片として検出ができると考える．

構文レベルではさらに制御文を要素とした制御文レベル，制御文に加え文の種類を含めた文レベル，文に加えて識別子を X とし演算子を含んだ式レベルを扱い，それぞれ抽象化を行う．構文レベルではさらに次の 3 つのレベルの抽象度によって抽象化を行う．

- 制御文レベル
- 文レベル
- 式レベル

制御文レベルは最も抽象度が高く，式レベルは最も抽象度が低い．3 つの抽象度を扱う理由は，類似している処理の中から例外的な処理を見つけることができるようにするためである．

名前レベルでは識別子を要素として扱い，そのままの識別子レベルと単語で区切った単語レベルでそれぞれ抽象化を行う．識別子だけでは似たような名前でも一文字でも異なっている場合に似ていないと判定されてしまうことがあり，意図した検出ができない場合がある．本研究では構成要素に構文レベルである制御文や文の種類，演算子を扱い，名前レベルでは識別子名や単語を扱い，類似度の算出を行うことでコードクローンでは検知できなかった類似したコード片を検出できると考える．

4.2.1 構文レベル

構文レベルでは編集距離を用いて類似度計算を行うので文字数の長さに意味を持たないように抽象化する必要があると考えた．制御文レベルでは制御文のみを抽出し，それぞれの制御文に対して抽象化を行う．抽象化のルー

ルは以下のように行う．

- if → I
- for → F
- while → W
- switch → S

文レベルでは制御文レベルに加えて文を残し，文の種類で抽象化を行う．抽象化のルールは以下のように行う．

- 関数呼び出し → A
- ジャンプ文 → J

ジャンプ文は return，continue，break， goto を対象としている．上記以外の文は抽象化せずセミコロンのみ残す．文の連接や入れ子関係の情報は保持される．式レベルでは文レベルに加えて演算子を残し，識別子を X として抽象化を行う．

4.2.2 名前レベル

名前レベルではコード片から識別子のみを抽出する．さらにキャメルケースやスネークケースを区切り，単語レベルとして抽象化を行う．

Listing5 は Listing1，Listing6 は Listing2 を抽象化したものである．

Listing 5 Listing1 を抽象化

```
1 I(){W()}
2 I(){W()A();}
3 I(!X){W(X.X<=X)A(X.X);}
4 E editor filerow Insert numrows Row row
```

Listing 6 Listing2 を抽象化

```
1 I(){I(){}}
2 I(){I(){A();J;}J;}
3 I(!X){I(X==X.X){A(X);JX;}J;}
4 E editor filerow fixcursor Insert numrows Row row
```

Listing5，Listing6 の 1 行目は制御文レベルで抽象化したもの，2 行目は文レベルで抽象化したもの，3 行目は式レベルで抽象化したもの，4 行目は単語レベルで抽象化したものである．

4.3 類似度の計算

本研究では 2 つのレベルで抽象化したコード片に対してそれぞれの類似度計算の手法を用いる．

4.3.1 構文レベル

構文レベルで抽象化したコード片に対しては要素の並びを考慮した編集距離であるレーベンシュタイン距離を用いた類似度計算を行う．レーベンシュタイン距離とは，2 つの文字列のうち 1 つの文字列に対して，1 文字の挿入・削除・置換によってもう一方の文字列に変形するために必要な手順の最小回数である．2 つの文字列 A, B の類似度 $sim(A, B)$ を求める式は次で表される．

$$sim(A, B) = 1 - \frac{\text{レーベンシュタイン距離}\ (A, B)}{max(A, B)}$$

レーベンシュタイン距離では文字列の長さを考慮しないので文字列の長さによって類似度が異なってくる場合がある．2 つの文字列の長い方で距離を割ることで標準化を行う．

4.3.2 名前レベル

名前レベルで識別子を抜き出したコード片に対しては並び順を考慮する必要がないとし Jaccard 係数を用いた類似度計算を行う．Jaccard 係数は 2 つの集合 A, B の類似性を表す指標で次の式で表される．

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

5 評価

評価の目的は提案する手法によってコメントのための類似処理が検出できているかどうかを確認することである．本章では 2 つの観点から評価を行う．1 つ目はコメントのための類似するコード片の検出が正しく行われているかどうか，2 つ目は提案する手法によって検出されるコード片について似ているかどうかについて評価を行う．実験対象は，kilo のソースファイル kilo.c を対象とし，コード片は if 文のみを対象とする．名前レベルでは識別子レベルを無視し単語レベルのみを扱う．提案手法の実現には Perl と Python を用いて実装を行なった．

5.1 コメント推薦箇所の評価

kilo.c から類似処理として Listing1 と Listing2 をグループ A とする．元のソースファイルの全体から算出された閾値を用いて，各レベルで類似しているかどうか判定し，コメント箇所推薦の箇所 1 が検出できるか評価を行う．閾値は元のソースファイルの類似度の平均に標準偏差を加算したものである．さらに箇所 2 を検出するために Listing3 と Listing4 をグループ B として用いる．表 1 は kilo.c の統計的なデータである．名前レベルでは低い数値になっていることがわかる．表 2 は各レベルの閾値である．閾値は各レベルの平均に標準偏差を加算したものである．

表 1 kilo の統計

	最大値	最小値	平均	中央値	分散	標準偏差
制御文レベル	1.0	0.064	0.634	0.6	0.099	0.315
文レベル	1.0	0.043	0.504	0.455	0.069	0.264
式レベル	1.0	0.033	0.314	0.299	0.022	0.149
名前レベル	1.0	0.0	0.045	0.0	0.01	0.1

表 2 閾値

	レベル	閾値
kilo.c	制御文レベル	0.949
	文レベル	0.768
	式レベル	0.463
	名前レベル	0.145

表 3 グループ A

	類似度
制御文レベル	0.7
文レベル	0.611
式レベル	0.571
名前レベル	0.875

表 4 グループ B

	類似度
制御文レベル	1.0
文レベル	0.917
式レベル	0.634
名前レベル	0.8

グループ A では構文レベルの制御文レベル，文レベルでは閾値を下回っているが，式レベルと名前レベルにお

表 5 コードの分類

No	制御文	文	式	名前	検出数
1	○	○	○	○	49
2	○	○	○	×	215
3	○	○	×	○	9
4	○	○	×	×	202
5	○	×	×	○	29
6	○	×	×	×	372
7	○	×	○	○	16
8	○	×	○	×	91
9	×	○	○	○	2
10	×	○	○	×	2
11	×	×	○	○	4
12	×	×	○	×	41
13	×	○	×	○	0
14	×	○	×	×	9
15	×	×	×	○	111
16	×	×	×	×	1539

いて閾値を超え，類似していると判定することができるので箇所 1 として検出することができる．グループ B では全てのレベルにおいて類似してると判定することができるので箇所 1 として検出することができる．構文レベルの式レベルにおいて閾値を下回る結果を期待していたが，閾値を超えているので箇所 2 として検出することができない．

5.2 提案手法によって検出されるコード片の評価

kilo.c を対象に提案手法によって検出されたコード片について評価する．kilo.c を対象を対象に提案手法の結果は表 5 となった．表 5 では kilo において分類されるパターンと検出されたコード片の組みの数を示す．コメント箇所推薦のための類似処理かどうかの判断は実際に検出されたソースコードを確認することで行う．検出された 16 種類のコード片の中から一つ無作為に選び類似しているかどうかの確認をする．似ている場合，コメントがついているかどうかの確認を行い，コメントがついている似ているコード片がある場合はコメント箇所推薦のための類似処理として分類する．コメントがない場合にはコメント箇所推薦のためではなく，類似処理として分類する．似ていない場合は類似処理ではないコード片として分類をする．

コメント箇所推薦のための類似処理として検出できるものは No1 から No3 と No5，No11 の合計 5 種類ある．類似処理でないものは No4，No6，No8，No10，No12 から No16 の合計 9 種類ある．類似処理だがコメント箇所推薦のためではないものが No7 と No9 の 2 種類ある．

6 考察

6.1 箇所 2 の検出

グループ B の例外処理は上手く検出されなかった．式レベルにおいて変数の値に 2 増やす処理と変数の値に 1 増やす処理で分類がされると予想を行ったが分類がされなかった．原因は条件式の中身が似ていることよって類似度が大きくなると考える．式レベルでは定数の違いによって分類がされると予想していたが，類似度計算において他の制御文や文の要素に大きく作用してしまうことで変化があまり見られなかった．++と+=2 の違いによって分類を行うためには条件式を除いた式文中の要素で類似度計算を行う必要がある．

本研究では名前レベルを用いることでコメント箇所推薦のための類似処理が検出できると考えたが，名前レベルによって類似していない場合でもコメント箇所推薦のための類似処理として検出できてしまうことや類似する場合でも類似処理でない場合がある．

6.2 類似処理の検出

本研究では構文レベルで抽象化した文字列の編集距離を用いて類似度を計算したが，木構造で表現して木の編集距離を用いた類似度計算も考えられる．木構造を保った状態で類似度の計算を行うことで，構造がより類似している処理を見つけることが可能になると考える．

7 おわりに

本研究では同一プロジェクト内のコメントのための類似したソースコード片の検出方法を提案し，評価を行なった．コメントのための類似したコード片とはコメントの整合性が取れていないコード片とその中で例外的な処理を行なっているコード片のことであり，制御構造や変数名などに着目した類似コード片の検出を行った．結果は箇所 1 について検出することができたが，箇所 2 については検出することができなかった．原因は，式レベルでは条件式によって類似判定がされてしまうと考える．

今後の課題として類似処理検出のための閾値について調整や，例外処理検出のための構成要素として新たなレベルを考える必要がある．

参考文献

[1] T. Kamiya, S. Kusumoto and K. Inoue, ”CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” in IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, July 2002

[2] A. Marcus and J. I. Maletic, ”Identification of high-level concept clones in source code,” Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), 2001, pp. 107-114

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, et al ”Comparison and Evaluation of Clone Detection Tools,” in IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 577-591, Sept. 2007.

[4] 神谷 年洋, 肥後 芳樹, 吉田 則裕, コードクローン検出技術の展開, コンピュータ ソフトウェア, 2011, 28 巻, 3 号, p. 3_29-3_42, 公開日 2011/09/26.

[5] 吉田，蜂巣，沢田，張ほか ”属性付き字句系列に基づくソースコード書き換え支援環境”，情報処理学会論文誌，Vol.53，No.7，pp.1832-1849，2012.

[6] Kilo，`https://github.com/antirez/kilo`