制御構造を考慮したソースコードコーパスに基づくメソッド呼び出し文補完手法

山本 哲男[1,a)]

概要：効率よくプログラムを作成するために既存のソースコードの再利用やライブラリを活用した開発が行われる．過去の研究において，同一メソッド内でよく利用される二つのメソッド呼び出し文の情報をソースコードコーパスとして保存し，そのコーパスをソースコード記述時に利用することでメソッド呼び出し文を補完する手法を提案した．本研究では，従来の手法を拡張することでより精度の高い補完手法を目指す．既存研究で作成したソースコードコーパスに制御文の有無情報を追加したソースコードコーパスを作成し，新たなソースコード解析の方法を提案する．さらに，提案手法を Eclipse プラグインとして実装し，従来手法と比較した評価実験についても述べる．

キーワード：コード補完，統合開発環境，コーパス

1. はじめに

ソフトウェアを開発する開発者は，フレームワークや既存のライブラリの API を組み合わせることでソースコードを記述していく．しかし，実現したい処理を取り扱うクラス名やメソッド名が分からないと，自ら処理を書く必要や一覧表などからクラスやメソッドを調べる時間が必要になり，手間がかかることになる．また，扱うクラスやメソッドが分かったとしても，メソッド呼び出し文を一文書けば完成するということは少なく，他の関連するメソッドや他クラスも利用してソースコードを組み上げていく必要がある．つまり，ソースコードを記述していく際には，API の呼び出し順序や次に必要になるであろうクラスやメソッドを知っておく必要がある．

そこで，既に記述されたソースコードや開発者が記述中のソースコードから情報を集めて解析し，適切な API やソースコードを推薦する仕組みに関する研究が多く存在する [1], [5], [7]．これらは，既存のソースコードの情報を利用することで，新規開発する開発者の求めているコード片や API を推薦する．[5] では API の呼び出し順を情報として利用し推薦する．[7] では二つの API 呼び出し間の関係を情報として利用して，メソッド呼び出し文自体の推薦を行う手法である．本研究では，[7] で提案している API 呼び出し間の関係をソースコードコーパスに保存して推薦する枠組みの拡張を行い，より推薦精度を上げる手法を提案する．拡張する方針は，二つの API 呼び出し間に存在する制御文に着目し，その情報もソースコードコーパスに追加することである．

```
1 String regex = "-(\\d+)";
2 String str = "Nihon-2014";
3 Pattern pattern = Pattern.compile(regex);
4 Matcher matcher = pattern.matcher(str);
5 if (matcher.find()){
6     String matchStr = matcher.group(1);
7 }
```

図 1 API を利用した Java ソースコード片

図 1 は Java で正規表現を利用し文字列を取得するソースコードであり，**Pattern** クラスや **Matcher** クラスを利用している．正規表現で文字列を取得するには，**Pattern.compile** メソッドを呼び出した後，**Pattern.matcher** メソッドを呼び出す．**Pattern.matcher** メソッドの呼び出しの返値として **Matcher** クラスのオブジェクトが取得できるので，そのオブジェクトを用い **Matcher.find** メソッドを呼び出す．その結果が真なら，正規表現で指定した文字列が存在するので **Matcher.group** メソッド呼び出しを行い，文字列を取得する．ここで重要なのは，**find** メソッドの呼び出し結果が真である時，**group** メソッドを利用できるところにある．**find** メソッドの結果を「if 文」で判定し，**group** メソッドを呼び出すという API の並びが典型的な処理となる．

[1] 日本大学工学部
College of Engineering, Nihon University
[a)] tetsuo@cs.ce.nihon-u.ac.jp

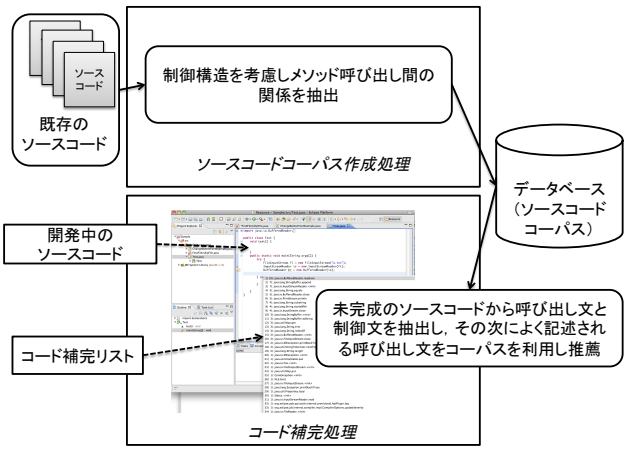

図 2 コード補完の流れ

そこで，API の呼び出し文間に制御文があるかどうかが重要であると考え，ある呼び出し文の後に if や while といった制御文がある場合とない場合で，推薦する呼び出し文を変更できれば，より適した呼び出し文が推薦できると考える．この考えを実現するためには，既存研究で作成したソースコードコーパスに制御文の有無情報を追加したソースコードコーパスを作成する必要が生じる．

本研究では，その拡張ソースコードコーパスとそれらの情報を取得し呼び出し文を開発者に提示する手法について提案する．この手法により，API ドキュメントを読むことで API を利用できる能力はあるが，その API の全体像を熟知していない開発者の手助けになると考える．

さらに，提案する手法を実装し，既存のオープンソースソフトウェアのソースコードを用いて実験を行った．制御文を考慮していない既存手法と考慮した提案手法で推薦する呼び出し文がどのように変化したかを計測し，提案手法の方がより上位に適切な結果が現れることを確認した．

以降，2 節で提案手法を説明し，3 節では実装したツールを用いて行った実験について述べる．4 節では，関連研究について触れ，最後に 5 節で本稿をまとめる．

2. 提案手法

2.1 補完手法の概要

本手法を利用した呼び出し文補完の全体の流れを図 2 に示す．本手法は「ソースコードコーパス作成処理」と，作成中の未完成なソースコードに対する「コード補完処理」に分けられる．

ソースコードコーパス作成処理では，あるメソッド呼び出し文の後に存在するメソッド呼び出し文として記述される文として，どのような文が多く記述されるかを既存のソースコードの中から抽出し，ソースコードコーパスに登録する．その際，その呼び出し間の間に制御構造の文が存在すれば，その情報も登録する．多くのソースコードを解析し，ソースコードコーパスに登録することで，ある呼び出し文の後に最もよく記述される呼び出し文の一覧が蓄えられる．

ただし，呼び出し文間の関係はその呼び出し文が記述されたメソッド内のみとする．また，if 文の then ブロックの呼び出し文と else ブロックの呼び出し文の間の関係といった，同時に実行されない文間の情報はコーパスに登録しない．

ソースコードコーパスに保存する情報は（前の呼び出し文，文間の制御キーワード，後の呼び出し文，出現数）の四つ組になる．「文間の制御キーワード」とは，文間の間に if 文が存在するのか，while が存在するのかといったキーワードの情報を表す．「文間の制御キーワード」についての詳細は 2.2 節で説明する．「出現数」の値は，（前の呼び出し文，文間の制御キーワード，後の呼び出し文）の組み合わせが既存のソースコードに出現した数を表す．

コード補完処理は，完成したソースコードコーパスからコード補完として必要な呼び出し文を取得する処理である．開発者はコード補完したいソースコード中の場所を統合開発環境等から指定する．指定した場所より前に記述されているソースコードの情報から次に記述されるであろう呼び出し文をソースコードコーパスから取得し開発者に提示する．

2.2 ソースコードコーパス作成処理

ソースコードコーパス作成の処理の流れは以下の通りである．

(1) ソースコードコーパスへ登録するソースコードの構文解析
(2) メソッド呼び出し文の呼び出しクラス・メソッドの解析
(3) ソースコード中のメソッド単位での呼び出しグラフの作成
(4) 呼び出しグラフの任意の頂点ペアについて実行経路が存在するか調べ，存在すれば頂点間の間に存在する制御キーワードを調査
(5) 頂点ペアと制御キーワードをソースコードコーパスに登録

ここで，呼び出しグラフとは今回提案する手法で用いるグラフである．頂点をメソッド呼び出し文と制御文に限定したフローチャートであり，以下の規則を適用し構築する．頂点集合はインスタンス生成文とメソッド呼び出し文を表す頂点（インスタンス生成文は<init>メソッド呼び出し文として扱い，メソッド呼び出し文と同等に扱う）と，制御キーワードを表す頂点（if，endif，do，enddo，while，endwhile，for，endfor の 8 種類）から構成される．制御文が存在しないメソッドの場合，呼び出し文を表す頂点を実行順に繋げたグラフとなる．

if 文が存在する場合，図 3 のように if 頂点から then ブ

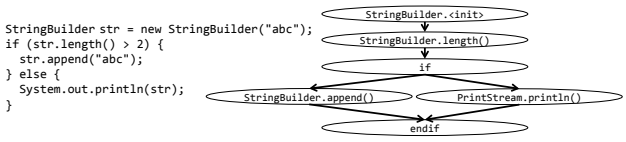

図 3 if 文を含むソースコードの呼び出しグラフ

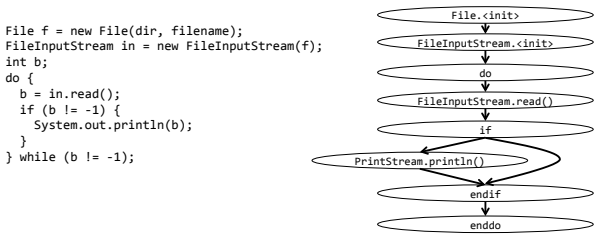

図 4 do 文を含むソースコードの呼び出しグラフ

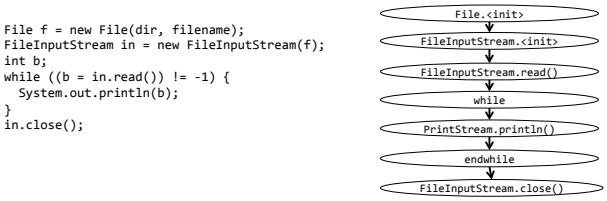

図 5 while 文を含むソースコードの呼び出しグラフ

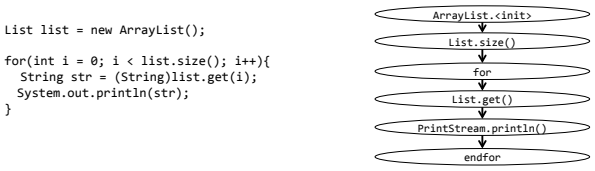

図 6 for 文を含むソースコードの呼び出しグラフ

ロックと else ブロックに分岐させ，endif 頂点で結合させる．ただし，if 文の条件式に記述された呼び出し文は if 頂点の直前に配置する．

do 文が存在する場合，図 4 のように do 頂点を配置し，do ブロックの内容を繋げ，enddo 頂点を最後に配置する．ただし，繰り返し判定の式に含まれる呼び出し文は enddo の直前に配置する．

while 文が存在する場合，図 5 のように while 頂点を配置し，while ブロックの内容を繋げ，endwhile 頂点を最後に配置する．ただし，繰り返し判定の式に含まれる呼び出し文は while の直前に配置する．

for 文が存在する場合，図 6 のように for 頂点を配置し，for ブロックの内容を繋げ，endfor 頂点を最後に配置する．ただし，for 文の初期化，繰り返し判定の式に含まれる呼び出し文は for の直前に配置し，更新処理に含まれる呼び出し文は endfor の直前に配置する．

たとえば，メソッドの内容が図 7 のソースコードであった場合は図 8 のような呼び出しグラフとなる．

すべてのメソッドの呼び出しグラフの作成が完了すると，グラフの任意の 2 頂点間（呼び出し文に限定）に対して，一方の頂点からもう一方の頂点へ到達可能か調べる．到達可能であった場合は，その到達経路中に存在する制御キーワードを表す頂点を列挙する．列挙するアルゴリズム

```
1  try {
2    ZipFile sourceZipFile = new ZipFile("E.zip");
3    String searchFileName = "readme.txt";
4
5    Enumeration e = sourceZipFile.entries();
6    boolean found = false;
7
8    System.out.println("Trying␣to␣search␣" +
         searchFileName);
9    while(e.hasMoreElements()) {
10     ZipEntry entry = (ZipEntry)e.nextElement();
11     if (entry.getName().toLowerCase().indexOf(
           searchFileName) != -1) {
12       found = true;
13       System.out.println("Found␣" + entry.
             getName());
14     }
15   }
16
17   if (found == false) {
18     System.out.println("File␣:" +searchFileName+
19       "␣Not␣Found␣Inside␣Zip␣File:␣" +
             sourceZipFile.getName());
20   }
21   sourceZipFile.close();
22 } catch(IOException ioe) {
23   System.out.println("Error␣opening␣zip␣file" +
         ioe);
24 }
```

図 7 ZIP ファイルを処理する Java ソースコード片

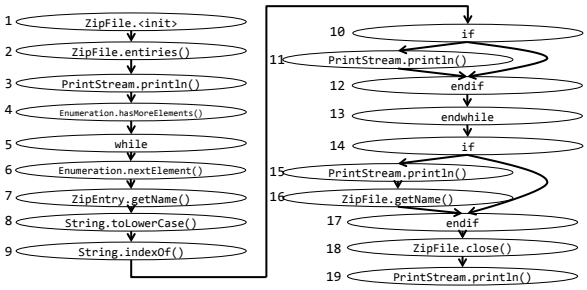

図 8 図 7 の呼び出しグラフ

入力：呼び出しグラフ
出力：制御キーワードを表す文字列

1: スタック *stack* を空に初期化
2: 呼び出し文の頂点をそれぞれ *src*，*dest* とする
3: **for all** *node* in *src* から *dest* への経路中の頂点 **do**
4: **if** *node* が if 頂点 **then**
5: “I” という文字を *stack* に push
6: **else if** *node* が while, do, for 頂点のいずれか **then**
7: “W” という文字 *stack* に push
8: **else if** *node* が endif, endwhile, enddo, endfor 頂点のいずれか **then**
9: **if** *stack* の top == *node* に対応する開始制御文 **then**
10: *stack* から pop し破棄
11: **end if**
12: **end if**
13: **end for**
14: *stack* に積まれた文字を bottom から並べた文字列が *src* と *dest* 間の「制御キーワード」

図 9 制御キーワード抽出アルゴリズム

入力：補完したい箇所を含むメソッドの呼び出しグラフ
出力：コード補完リスト

1: 呼び出しグラフから補完したい箇所の直前の頂点を特定（頂点 $dest$ とする）．
2: 頂点 $dest$ からグラフを逆向きにたどり到達するすべての呼び出し文頂点を列挙（列挙した集合を頂点集合 S とする）．
3: **for all** src in 頂点集合 S **do**
4: 　src から頂点 $dest$ の間の制御キーワード $keyword$ を図 9 のアルゴリズムで調べ，$(src, keyword)$ のペアを作成する．
5: 　$(src, keyword)$ の情報を元にソースコードコーパスに問い合わせ，$(src, keyword, destcandidate, count)$ の四つ組みがあるか調査（$destcandidate, count$ は任意）．
6: 　$destcandidate$ と $count$ を取得する．複数ある場合はすべて取得し，$count$ の合計値 sum を計算する．
7: 　$(src, destcandidate, count/sum)$ の三つ組を作成し，候補集合 C に追加する．
8: **end for**
9: **while** 候補集合 C が空でない **do**
10: 　候補集合 C から三つ組を一つ取り出し，C から取り除く（c とする）
11: 　2 番目の要素（$destmethod$ とする）が c と同じ三つ組を候補集合 C から探す（結果を集合 D とする）
12: 　集合 D のすべての要素の 3 番目の要素の合計値を $total$ とする
13: 　$(destmethod, total)$ ペアをコード補完リストに追加
14: **end while**
15: コード補完リストを $total$ で整列

図 10　コーパス問い合わせアルゴリズム

を図 9 に示す．呼び出し文間のネストの深さに応じて制御キーワードの長さが変わることになる．たとえば，図 8 の 2 番目の頂点と 11 番目の頂点間の制御キーワードは “WI” となる．間に while 文と if 文があるからである．一方，2 番目と 18 番目の頂点間の制御キーワードは空文字列である．間に while 文等が存在するが，すべて end 頂点で閉じているため，2 頂点間には制御関係がないものと考え，空となる．

なお，呼び出し文の引数は扱わない．そのため，引数の種類や数だけが異なるメソッド呼び出し文は同じ呼び出し文として処理する．また，try・catch 文といった例外処理による制御は無視し，ソースコード上に出現した順で呼び出し文を繋げて呼び出しグラフを作成する．

2.3　コード補完処理

ソースコードコーパスからコード補完のために必要な情報を取得する方法について説明する．最初に，候補に必要な情報を統合開発環境上のソースコードから取得する必要がある．開発者は，利用中の統合開発環境上で編集中のソースコードとそのソースコード上の補完したい箇所を明示する．明示後からコード補完リストの作成の流れを図 10 に示す．最終的に開発者に提示される情報はランク付けされた呼び出し文のリストであり，コード補完リストと呼ぶ．

コード補完リストを見た開発者は，リストを参考に開発を継続する．そのリストから必要と思われる呼び出し文を選び記述すればよい．ただし，その呼び出し文を記述する場合にはオブジェクトへの参照が必要な場合があるため，オブジェクトの参照を格納している変数などを用いて開発者自らメソッド呼び出し文を記述する必要がある．

3.　評価

3.1　適用例

2010 年 5 月 22 日にチェックアウトした Eclipse のソースコードを用いてソースコードコーパスを作成した．Java ファイルの総数は 60,265 個，総行数（空行，コメント行を含む）は 10,083,198 である．

作成したコーパスを用いて，図 1 のソースコードでコード補完を利用した例について説明する．図 1 の 5 行目の呼び出し文の後でコード補完をしたいと思った場合を例とする．それぞれの箇所からソースコードコーパスに候補リストを問い合わせると，表 1 に示す結果が得られる．表中の順位の右に記述されている括弧内の値は図 10 の $total$ を表す．ただし，6 位以下の候補は省略している．また，5 行目が if 文ではなく，**matcher.find** メソッド呼び出し文だけとした場合の結果を表 2 に示す．いずれも補完を指定した地点の次の呼び出し文がコード補完リストの 1 位となっている．

同様な測定を図 7 に示すソースコードに対しても行った．表 3 は図 7 で 89 行目の直後で補完を実施した場合のコード補完リストの結果を示す．なお，本手法の結果だけでなく従来手法の結果も併せて示す．また，補完を実施した直後の呼び出し文が現れる行には下線を引いてある．

表 3 の結果を見ると，従来手法では 4 番目に存在し，本手法では 1 番目に存在し，従来手法より上位にランキングされていることが分かる．

3.2　有効性の評価

既存のソースコードを利用して，コード補完リストがどの

表 1　図 1 の 5 行目直後でのコード補完リスト

順位 (値)	呼び出し文
1(34)	java.util.regex.Matcher.group
2(21)	java.util.regex.Matcher.start
3(18)	java.lang.String.substring
4(12)	java.lang.String.length
5(7)	java.util.ArrayList.add
5(7)	java.util.regex.Matcher.find

表 2　図 1 の 5 行目の if がない場合のコード補完リスト

順位 (値)	呼び出し文
1(30)	java.util.regex.Matcher.find
2(22)	java.util.regex.Pattern.matcher
3(14)	java.lang.String.length
4(7)	java.util.regex.Matcher.group
5(7)	java.util.regex.Matcher.matches

入力：メソッドとそのメソッド内の何番目の呼び出し文かを表す数 n
出力：n 番目の呼び出し文の直後の呼び出し文が補完可能かどうか

```
1: if メソッド中の呼び出し文の数 >= n + 1 then
2:     if n + 1 番目の呼び出し文が自アプリケーション内のクラスの呼び出し文かどうか then
3:         「内部呼び出し数」を一つ増やす
4:     else
5:         n 番目の呼び出し文の直後を補完したい箇所としてソースコードコーパスへ問い合わせし，コード補完リストを取得
6:         if コード補完リスト中に n + 1 番目の呼び出し文が含まれるかどうか then
7:             「コード補完リストに存在する数」を一つ増やす
8:         else
9:             「コード補完リストに存在しない数」を一つ増やす
10:        end if
11:    end if
12: end if
```

図 11 計測手順

程度有効かについて計測をした結果について述べる．ソースコードコーパスは 3.1 節で作成したものを用い，コード補完リストを表示させるためのソースコードは Eclipse Plugin の一つである CheckstylePlugin 5.5.0[*1] のソースコードを利用した．このプラグイン内の Java ファイルの総数は 188 個であり，総行数は 37699 行である．

188 個あるファイル中のすべてのクラスのメソッドに対して，「コード補完リストに存在する数」，「コード補完リストに存在しない数」，「内部呼び出し数」，という 3 種類の数を計測する．「内部呼び出し数」とはアプリケーション内の別のメソッド呼び出し文をメソッド内に記述している場合を指す．

計測方法を図 11 に示す．1 以上の整数である n を設定し，すべてのメソッドに対して図 11 に示す手順を実行する．自アプリケーション内のクラスの呼び出し文かどうかの判定は，本評価の場合，CheckstylePlugin のソースコード内のクラスかどうかで判定する．

従来手法で n を 5, 10, 15, 20, 25 として測定した結果を表 4 に示し，本手法で測定した結果を表 5 に示す．存在する数の右の括弧は，「存在する数」＋「存在しない数」中に「存在する数」の割合を示している（正解数の割合と呼ぶ）．また，コード補完リストに存在する数の中で何番目にランキングされているかを計測した結果を表 6 と表 7 に示す．上位 5 位以内，10 位以内，15 位以内，20 位以内としてまとめており，括弧内の数字は割合を示している．なお，「存在する数」と「存在しない数」と「内部呼び出し数」の和が従来手法と本手法で異なるのは，本手法から内部クラスを解析対象に含めたためである．

本手法で，n が 5 の時にコード補完リストに存在するメソッドは 147 個存在し，存在しなかったメソッドは 21 個であった．また，内部の呼び出し文のメソッドは 203 個であったことが分かる．そして，147 個の内訳として，上位 5 位以内に呼び出し文が含まれているメソッドは 117 個あり，10 位以内は 127 個あったことが分かる．

表 3 図 7 の 8 行目直後でのコード補完リスト (上段：従来手法，下段：本手法)

順位 (値)	呼び出し文
1(45)	java.io.PrintStream.println
2(8)	java.util.zip.ZipEntry.getName
3(7)	java.util.zip.ZipFile.close
4(7)	**java.util.Enumeration.hasMoreElements**
5(7)	java.util.Enumeration.nextElement
1(31)	**java.util.Enumeration.hasMoreElements**
2(9)	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpression
3(7)	java.util.zip.ZipFile.getEntry
4(6)	java.util.zip.ZipFile.close
5(6)	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpressionWithName

[*1] http://eclipse-cs.sourceforge.net/

3.3 考察

本手法は制御文の有無で補完候補が変わる手法である．表 1 は if 文の直後にくる呼び出し文の候補一覧であるが，

表 4 CheckstylePlugin を用いた実験結果（従来手法）

n	存在する数	存在しない数	内部呼び出し数
5	117(78.0%)	33	201
10	53(84.1%)	10	117
15	36(83.7%)	7	67
20	30(88.2%)	4	33
25	19(100.0%)	0	19

表 5 CheckstylePlugin を用いた実験結果（本手法）

n	存在する数	存在しない数	内部呼び出し数
5	147(87.5%)	21	203
10	81(92.0%)	7	108
15	54(88.5%)	7	61
20	21(84.0%)	4	54
25	19(90.5%)	2	26

表 6 候補リストの上位に含まれる数（従来手法）

n	5 位以内	10 位以内	15 位以内	20 位以内
5	65(55.6%)	76(65.0%)	86(73.5%)	89(76.1%)
10	25(47.1%)	33(62.3%)	36(67.9%)	37(69.8%)
15	25(69.4%)	26(72.2%)	28(77.8%)	28(77.7%)
20	17(56.6%)	19(63.3%)	20(66.7%)	20(66.7%)
25	11(57.9%)	14(73.7%)	15(78.9%)	16(84.2%)

表 7 候補リストの上位に含まれる数（本手法）

n	5 位以内	10 位以内	15 位以内	20 位以内
5	117(79.6%)	127(86.4%)	129(87.8%)	131(89.1%)
10	41(50.6%)	64(79.0%)	67(82.7%)	67(82.7%)
15	34(63.0%)	37(68.5%)	40(74.1%)	41(75.9%)
20	15(71.4%)	18(85.7%)	19(90.5%)	19(90.5%)
25	12(63.2%)	13(68.4%)	13(68.4%)	15(78.9%)

もし，if が記述されていなかった場合は表 2 のような結果になる．if の存在で候補が変わり，適切な呼び出し文が上位にくることが分かる．

従来手法と比較した結果である表 3 をみると，いずれの場合も従来手法に比べて上位にランキングされていることが分かる．

これらの結果から，制御構造を考慮することで，より精度の高い推薦が行えると考える．実験の例で利用したサンプルソースコードのように典型的な API 呼び出し文に対してはコード補完リストが有効となる．本手法の利点は必要なクラス名が分からなくても，クラス名とメソッド名を提示してくれるところにある．もちろん，クラス名が分かっていた場合にも候補の表示は可能である．

3.2 節のオープンソースソフトウェアを利用した比較実験では，n が 5, 10, 15 のときは候補に含まれる正解数の割合が上昇した．逆に，n が 20, 25 のときは下降した．提案手法の場合，n の数がどのような数でも，8 割以上の確率で正解呼び出し文が候補一覧に含まれることが分かる．さらに，表 6 と表 7 を比較すると，n が 5, 10, 20 のときに，本手法の方が上位に正解が来る傾向にある．しかし，n が 10 のときは従来手法の方が上位に正解が集まっている．

ただし，これらの実験結果はソースコードコーパスの内容に依存する可能性がある．今回は Eclipse のソースコードを用いて，プラグインに関係する処理を評価したが，異なるドメインのソフトウェアを利用した場合は結果が大幅に異なる可能性がある．また，本手法においても，従来手法と同様にコーパスに存在しないクラスの情報は候補リストに含める事が出来ない問題は発生する．

4. 関連研究

API に着目しコード検索を行う手法として Mishne らの手法 [5] がある．この手法では，API の実行順に着目し，その API の並びをクエリーとしてコード検索を行う．コード検索のための手法であるため，そのままでは補完に適さない．

API の情報を既存のソースコードから抽出し，役立たせる研究は数多く存在する．Prospector[4] や Xsnippet[6] は，あるメソッドの返り値を利用して，さらにメソッドを呼び出すといった連鎖がある場合に，その情報をデータベースに入れておきコードアシストをするツールである．

Strathcona[2], [3] は，ソースコードの構造に基づき，コード例を推薦してくれるツールである．ただし，コード例を提示するシステムであり，文単位での推薦はしてくれない．

既存のソースコードを利用したコード補完手法に Bruch ら [1] の手法がある．この手法は，あらかじめフレームワークなどでオーバーライドして記述するメソッド内でよくつかわれるメソッド一覧を既存のソースコードから作成しておく．そして，そのフレームワークを利用してオーバーライドするメソッドを開発する際に，そのメソッド内で最適なメソッドを提示してくれるものである．本手法は，メソッドの種類を限定せずあらゆる場面で利用可能な点が異なる．また，我々の手法はメソッド名だけでなく，メソッド呼び出し文全体を補完することが可能である．

5. おわりに

本稿では，メソッド呼び出し分とその間の制御文に着目し，メソッド呼び出し文を補完する手法について提案した．事前にソースコードコーパスとして任意のメソッド呼び出し文の後に存在するメソッド呼び出し文とその間に存在する制御キーワードを記録しておき，その情報を用いて，適切なメソッド呼び出し文を開発者に提示する．さらに，これらの手法を Eclipse プラグインとして実装し，従来研究との比較実験を行った．実験の結果，候補の上位により適切なメソッド呼び出し文が存在することを確認した．

今後は，同一メソッド内のメソッド呼び出し文だけでなく，メソッドにまたがった呼び出し文の構造も考慮することで，より精度の高い推薦結果を求めることが挙げられる．

謝辞 本研究は栢森情報科学振興財団の助成を受けて遂行された．

参考文献

[1] Bruch, M., Monperrus, M. and Mezini, M.: Learning from examples to improve code completion systems, *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, New York, USA, ACM, pp. 213–222, (2009).

[2] Holmes, R. and Murphy, G. C.: Using structural context to recommend source code examples, *Proceedings of the 27th international conference on Software engineering - ICSE '05*, New York, New York, USA, ACM Press, pp. 117–125, (2005).

[3] Holmes, R., Walker, R. and Murphy, G.: Approximate Structural Context Matching: An Approach to Recommend Relevant Examples, *IEEE Transactions on Software Engineering*, Vol. 32, No. 12, pp. 952–970, (2006).

[4] Mandelin, D., Xu, L., Bodik, R. and Kimelman, D.: Jungloid mining: helping to navigate the API jungle, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Vol. 40, No. 6, ACM, pp. 48–61, (2005).

[5] Mishne, A., Shoham, S. and Yahav, E.: Typestate-based Semantic Code Search over Partial Programs, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, New York, NY, USA, ACM, pp. 997–1016, (2012).

[6] Sahavechaphan, N. and Claypool, K.: XSnippet: mining For sample code, *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Vol. 41, No. 10, ACM, pp. 413–430, (2006).

[7] 山本哲男：ソースコードコーパスを利用したメソッド呼び出し文補完手法，情報処理学会論文誌，Vol. 54, No. 2, pp. 903–911 (2013).