情報処理学会論文誌 Vol.56 No.2 682–691 (Feb. 2015)

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)]

受付日 2014年5月19日, 採録日 2014年11月10日

概要：効率良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 2 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

A Code Completion of Method Invocation Statement Using Source Code Corpus including a Consideration of Contol Flow

Tetsuo Yamamoto[1,a)]

Received: May 19, 2014, Accepted: November 10, 2014

Abstract: Programmers reuse existing source code or use libraries to develop effectively. There has been past work on code completion. In the work, source code corpus stores information on two method invocation statements in a method in advance, and then programmers get information on code completion from the corpus. This study aims to improve the degree of precision to extend the work. We have made novel source code corpus including a consideration of control flow and propose a source code analysis to make it. We have implemented the approach as Eclipse Plugin, and describe the approach is effective through comparative evaluations with the past work.

Keywords: code completion, integrated development environment, corpus

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，すで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], [7], [10]．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．文献 [7] では API の呼び出し順を情報として利用し推薦する．文献 [10] では 2 つの API 呼び出

[1] 日本大学工学部
College of Engineering, Nihon University, Koriyama, Fukushima 963–8642, Japan
[a)] tetsuo@cs.ce.nihon-u.ac.jp

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，文献 [10] 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，2 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．

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とき，**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．

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 章で従来手法の概要および問題点を述べ，さらに制御構造を利用するに至った動機を説明する．3 章で提案手法を説明し，4 章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．5 章では，関連研究について触れ，最後に 6 章で本稿をまとめる．

```
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
Fig. 1 Java source code snippet using API.

2. 従来手法

本章では，文献 [10] で提案している「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」について説明し，その問題点を述べる．

2.1 従来手法の概要

補完手法の流れを**図 2** に示す．処理の流れはソースコードコーパス作成処理と，開発中のソースコードに対するコード補完処理に分けられる．なお，本手法で説明するソースコードは Java で記述されているものとする．

ソースコードコーパス作成処理では，既存のソースコードを解析し，その情報をデータベース（ソースコードコーパスと呼ぶ）に保存する処理が必要になる．はじめに，ソースコードを解析し呼び出し文を抽出する．その後，メソッド単位で呼び出し文を順番にならべ，文と文の組み合わせをソースコードコーパスへ保存する．図 1 のソースコードを既存のソースコードとした場合を考えると，ソースコード中に 4 つの呼び出し文が存在するので，6 通りの組合せになる．そのすべての組み合わせを保存する．

保存する情報は（前の呼び出し文，後の呼び出し文，出現数）の三つ組とする．すでに，保存しようとする組合せがソースコードコーパス内に存在した場合は，出現数の値を 1 増やし更新する．なければ，出現数を 1 として保存する．これらの処理をソースファイル中のすべてのメソッドに対して実行する．ソースコードコーパスの作成は一度作成すればよく，ソースコードを追加したい場合は，既存のソースコードコーパスへ解析した情報を保存するだけですむ．

コード補完処理では，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．コード補完したいソースコード中の場所は開発者が指定するものとする．

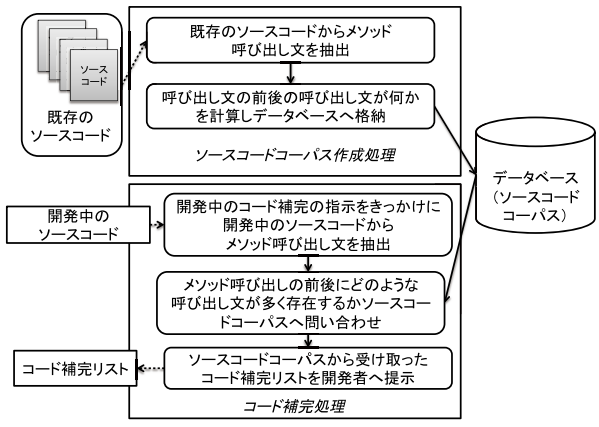

図 2 コード補完の流れ
Fig. 2 A flow of code completion.

表 1 図 1 の 5 行目のコード補完リスト
Table 1 Code completion list after line 5 in Fig. 1.

順位	呼び出し文
1	java.util.regex.Matcher.find
2	java.util.regex.Pattern.matcher
3	java.lang.String.length
4	java.util.regex.Matcher.group
5	java.util.regex.Matcher.matches

まず，指定した場所を含むメソッド内にすでに記述されている呼び出し文を抽出する．取得した呼び出し文をキーとしてソースコードコーパスに問い合わせ，そのキーとなる呼び出し文の後に出現する呼び出し文の一覧を取得する．この処理を取得した呼び出し文のすべてに対して実施し結果を統合する．最後に，出現数の多い順に並べ替えて開発者に提示する．このリストをコード補完リストと呼ぶ．

2.2 従来手法の問題点

図 1 の 5 行目まで記述し，コード補完処理を実施すると**表 1** の結果が得られる．期待する呼び出し文はコード補完リストの 4 位に存在する．1 位は直前の呼び出し文と同じ **Matcher.find** メソッド呼び出し文となっている．理由としては，さらにその前の呼び出し文である **Pattern.compile** メソッド呼び出しと **Pattern.matcher** メソッドを呼び出しの後方に **Matcher.find** 呼び出しが現れる確率が高く，統合された補完リストにその影響が反映されるためである．

そこで，本研究では，ソースコード中に出現した呼び出し文の並びだけではなく，その呼び出し文の間にある制御キーワードも考慮してソースコードコーパスを作成し，補完リストを生成する手法を提案する．

たとえば，**Pattern.compile** メソッド呼び出しや **Pattern.matcher** メソッド呼び出し後に if 文が出現し，その if 文の後に現れる呼び出し文としてどのような呼び出し文が多く現れるかをソースコードコーパスに保存する．そのように保存する情報を改良し，コード補完リストを生成することで，より上位に **Matcher.find** メソッド呼び出し文がランキングされるようにする．

3. 提案手法

本章では，提案するメソッド呼び出し文補完手法について説明する．

基本的な流れは従来手法と同様だが，以下の点で手法を改良する必要がある．

- 任意の呼び出し文間に制御構造があるかどうかを調べる．
- ソースコードコーパスに制御構造を含めたを情報を保存する．

以降，改良した手法を用いた全体の流れを説明し，次に，ソースコードコーパスの作成手法，統合開発環境からソースコードコーパスを利用する手法について説明する．最後に，それらの手法の実装に関して述べる．

3.1 補完手法の概要

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 3.2 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）の組合せが既存のソースコードに出現した数を表す．

次に，コード補完処理について説明する．たとえば，図 1 のソースコードで 3 行目までしか記述していないとする．そして，開発者が 3 行目の後でコード補完をしたいと思った場合を想定する．このとき，統合開発環境では，指定位置の前に存在する呼び出し文として，3 行目の **Pattern** クラスの **compile** メソッド呼び出し文を抽出する．さらに，補完したい場所と抽出したい呼び出し文の間に制御キーワードがないため，「文間の制御キーワード」は空文字とする．これらの情報をソースコードコーパスに問い合わせ，**Pattern** クラスの **compile** メソッドの後に書かれるであろう呼び出し文の一覧を取得する．最後に，一覧を出現数の降順に並べ替え開発者に提示する．

また，図 1 のソースコードで 5 行目までしか記述していない場合も考える．さきほどと同様に指定位置の前に存在する呼び出し文として，3 行目の **Pattern** クラスの **compile** メソッド呼び出し文，4 行目の **Matcher** クラスの **matcher** メソッド呼び出し文と 5 行目の **Matcher** クラスの **find** メソッド呼び出し文を抽出する．指定位置から各呼び出し文の間の制御キーワードをみると，if があるため，「文間の制御キーワード」を「if」としてソースコー

ドコーパスに問い合わせ，取得した一覧を統合し提示する．統合する際に出現数の情報を出現割合に変換し統合する．

3.2 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とおりである．

(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・意味解析
(2) 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
(3) 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
(4) 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つなげたグラフとなる．

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 ブロックの内容をつなげ，enddo 頂点を最後に配置する．

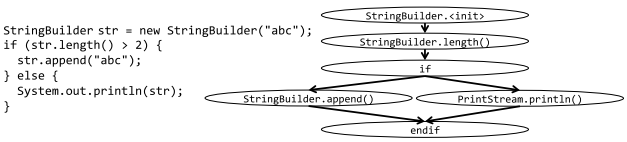

図 3 if 文を含むソースコードの呼び出しグラフ

Fig. 3 Call statements grpah of source code including an if statement.

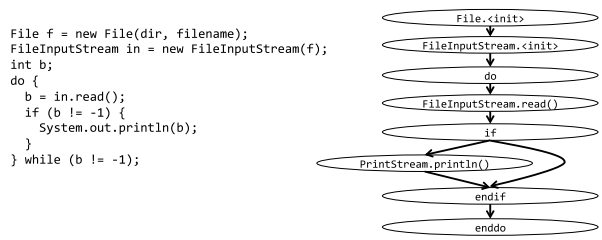

図 4 do 文を含むソースコードの呼び出しグラフ

Fig. 4 Call statements grpah of source code including a do statement.

ただし，繰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 ブロックの内容をつなげ，endwhile 頂点を最後に配置する．ただし，繰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 ブロックの内容をつなげ，endfor 頂点を最後に配置する．ただし，for 文の初期化，繰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．

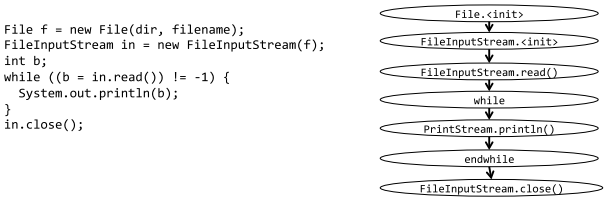

図 5 while 文を含むソースコードの呼び出しグラフ

Fig. 5 Call statements grpah of source code including a while statement.

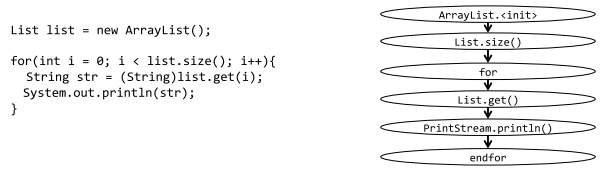

図 6 for 文を含むソースコードの呼び出しグラフ

Fig. 6 Call statements grpah of source code including a for statement.

```
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

Fig. 7 Java source code snippet to search a file in ZIP file.

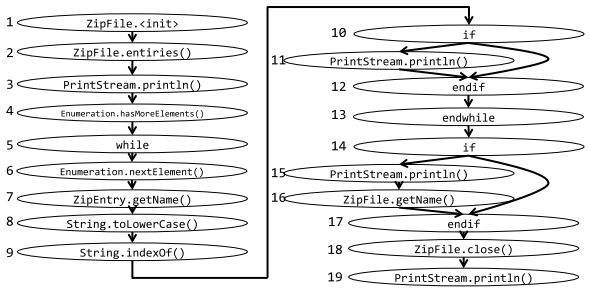

図 8 図 7 の呼び出しグラフ
Fig. 8 Call statements graph of Fig. 7.

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
Fig. 9 Algorithm of corpus query.

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 文などが存在するが，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つなげて呼び出しグラフを作成する．

3.3 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．

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
Fig. 10 Algorithm of corpus query.

既存のソースコードに記述された呼び出し文の並びで最も出現確率が高いものが候補として有用という考えに基づき，コード補完リストを作成する．補完したい箇所より前に記述された呼び出し文と制御キーワードをキーとして，ソースコードコーパスに問い合わせる．その呼び出し文の後に出現する呼び出し文として出現確率が高い呼び出し文を取得することで，コード補完リストを作成する．

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．

たとえば，図 7 の 10 行目の先頭で補完を実施したときを考える．このとき，頂点 *dest* は図 8 の 5 番頂点となり，頂点集合 S は {**ZipFile.<init>**, **ZipFile.entries**, **PrintStream.println**, **Enumeration.hasMoreElements**} となる．それぞれの制御キー

ワードは “W” となる．**表 2** は各頂点の情報をソースコードコーパスに問い合わせた結果を示す．ただし，割合に変換しソート後の上位三件を表している．最後に，これらの表を統合することでコード補完リストが完成する．結果を表 9 の下段に示す．

3.4 実装

本節では本手法を実装したツールについて説明する．開発者が利用する統合開発環境は Eclipse とし，ソースコード解析には Eclipse JDT（Java Development Tools）[*1]の解析器を利用している．ソースコードコーパスとして実装するデータベースには BerkleyDB [*2]を用いている．これらの実装は従来研究 [10] の実装を拡張したものである．

Java のソースコードを解析するにあたって従来の実装では内部クラスを取り扱っていなかったが，今回の実装から内部クラスも解析対象に含め，内部クラスのメソッドも解析を行う．解析対象を増やすことで，1 つの Java ソースファイルからより多くの情報をソースコードコーパスに追加できるようになる．

また，ソースコードコーパスに四つ組を登録するときに，四つ組の 1 番目の要素と 2 番目の要素を結合し，キーにすることで，問い合わせ時の処理の高速化を図っている．こうすることで，呼び出し文と制御キーワードを用いたキーを作成し，ソースコードコーパスからリストを取得するだけになり，高速な処理が可能となる．

表 2 ソースコードコーパスへの問い合わせ結果
Table 2 A result of querying the corpus.

java.util.Enumeration.hasMoreElements, “W”

順位（割合）	呼び出し文
1(0.15)	java.lang.String.substring
1(0.15)	java.util.Enumeration.nextElement
3(0.12)	java.util.zip.ZipEntry.getName

java.io.PrintStream.println, “W”

1(0.11)	java.io.PrintStream.println
2(0.04)	java.util.Iterator.next
3(0.02)	java.io.PrintStream.print

java.util.zip.ZipFile.entries, “W”

1(0.20)	java.util.Enumeration.nextElement
2(0.19)	java.util.zip.ZipEntry.getName
3(0.07)	java.util.zip.ZipEntry.isDirectory

java.util.zip.ZipFile.<init>, “W”

1(0.24)	java.util.Enumeration.nextElement
2(0.18)	java.util.zip.ZipEntry.getName
3(0.09)	java.util.zip.ZipEntry.isDirectory

[*1] http://www.eclipse.org/jdt/
[*2] http://www.oracle.com/technetwork/database/berkleydb/overview/index.html

4. 評価

提案手法を評価するために実験を行った．本章では，実用的な速度で実現可能かを検証するためのパフォーマンス評価について記述し，その後，本手法の有効性の評価について述べる．

4.1 データベースのパフォーマンス評価

ソースコードコーパスとなるデータベースの構築に要した時間と構築したデータベースのサイズについて調査した．この調査では，Eclipse を対象として調査した．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た[*3]．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．これらには，Eclipse 本体のソースコードはもちろん，テスト用のソースコードも含まれる．

サーバの計算機として CPU が Core i5 3.2 GHZ でメモリが 32 GB の計算機を利用したところ，構築時間は約 50 分かかり，データベースのサイズは 5.7 GB 必要になった．

ソースコードの登録処理は，利用開始前の一度だけ行えばよく，利用開始後は更新されたファイルについてのみデータベースを更新をすればよいため，登録時のパフォーマンスは問題ないと考える．

4.2 適用例

4.1 節で作成したソースコード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 3 行目，4 行目，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 3**，**表 4**，**表 5** に示す結果が得られる．ただし，11 位

表 3 図 1 の 3 行目直後でのコード補完リスト
Table 3 Code completion list after line 3 in Fig. 1.

順位 (値)	呼び出し文
1(11)	java.util.regex.Pattern.matcher
2(9)	java.util.regex.Matcher.find
3(3)	java.util.regex.Pattern.compile
4(3)	java.util.ArrayList.iterator
5(3)	java.util.Iterator.hasNext
6(2)	SearchPattern.<init>
7(2)	NLS.bind
8(2)	org.eclipse.ui.tests.dialogs.SearchPatternAuto.assertEquals
9(2)	org.eclipse.pde.internal.core.util.PatternConstructor.asRegEx
10(2)	org.eclipse.ui.tests.dialogs.SearchPattern.setPattern

[*3] http://2011.msrconf.org/msrc2011/eclipse-cvs.tgz をダウンロード・展開し，指定した日付でチェックアウトしたもの

表 4 図 1 の 4 行目直後でのコード補完リスト
Table 4 Code completion list after line 4 in Fig. 1.

順位 (値)	呼び出し文
1(23)	java.util.regex.Matcher.find
2(16)	java.util.regex.Pattern.matcher
3(7)	java.util.regex.Matcher.matches
4(5)	java.util.regex.Pattern.compile
5(5)	java.lang.String.length
6(4)	java.util.Iterator.hasNext
7(3)	java.lang.String.substring
8(3)	NLS.bind
9(3)	java.util.ArrayList.iterator
10(2)	org.eclipse.ui.tests.dialogs.SearchPatternAuto.assertEquals

表 5 図 1 の 5 行目直後でのコード補完リスト
Table 5 Code completion list after line 5 in Fig. 1.

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
6(7)	java.util.regex.Matcher.find
7(5)	org.eclipse.jface.text.Region.<init>
8(5)	java.util.regex.Matcher.end
9(4)	java.lang.Integer.parseInt
10(4)	org.eclipse.e4.ui.workbench.swt.internal.RGB.toString

表 6 図 1 の 5 行目の if がない場合のコード補完リスト
Table 6 Code completion list in case of no if statement on line 5 in Fig. 1.

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
6(6)	java.util.regex.Pattern.compile
7(6)	java.util.regex.Matcher.reset
8(5)	java.lang.String.substring
9(5)	java.lang.StringBuffer.toString
10(5)	java.util.regex.Matcher.start

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 6** 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 7**，**表 8**，**表 9** はそれぞれ図 7 で 5，8，9 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あわ

表 7 図 7 の 5 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）
Table 7 Code completion list after line 5 in Fig. 7 (Upper: past method, Bottom: this method).

順位 (値)	呼び出し文
1(8)	java.util.zip.ZipEntry.getName
2(7)	java.util.zip.ZipFile.close
3(7)	java.util.Enumeration.hasMoreElements
4(7)	java.util.Enumeration.nextElement
5(4)	java.io.File.<init>
6(4)	java.util.zip.ZipFile.getInputStream
7(4)	java.lang.String.length
8(3)	java.util.zip.ZipFile.getEntry
9(3)	java.lang.String.substring
10(3)	java.io.InputStream.close
1(31)	java.util.Enumeration.hasMoreElements
2(7)	java.util.zip.ZipFile.getEntry
3(6)	java.util.zip.ZipFile.close
4(3)	java.util.zip.ZipFile.entries
5(2)	ImportOperation.<init>
6(2)	CoreException.<init>
7(2)	java.util.HashSet.add
8(2)	Path.<init>
9(2)	Status.<init>
10(2)	java.util.ArrayList.<init>

表 8 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）
Table 8 Code completion list after line 8 in Fig. 7 (Upper: past method, Bottom: this method).

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
6(5)	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpression
7(4)	java.io.File.<init>
8(4)	java.util.zip.ZipFile.getInputStream
9(4)	java.lang.String.length
10(3)	java.util.zip.ZipFile.getEntry
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
6(3)	java.util.zip.ZipFile.entries
7(2)	ImportOperation.<init>
8(2)	CoreException.<init>
9(2)	org.eclipse.jdt.internal.compiler.parser.Parser.consumeAssignmentOperator
10(2)	java.util.HashSet.add

表 9 図 7 の 9 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）
Table 9 Code completion list after line 9 in Fig. 7 (Upper: past method, Bottom: this method).

順位 (値)	呼び出し文
1(45)	java.io.PrintStream.println
2(20)	**java.util.Enumeration.nextElement**
3(8)	java.util.zip.ZipEntry.getName
4(8)	java.util.Enumeration.hasMoreElements
5(7)	java.util.zip.ZipFile.close
6(5)	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpression
7(5)	java.lang.String.length
8(4)	java.lang.String.substring
9(4)	java.io.File.<init>
10(4)	java.util.zip.ZipFile.getInputStream
1(60)	**java.util.Enumeration.nextElement**
2(49)	java.util.zip.ZipEntry.getName
3(22)	java.util.zip.ZipEntry.isDirectory
4(16)	java.lang.String.substring
5(11)	java.io.PrintStream.println
6(11)	java.lang.String.endsWith
7(10)	java.lang.String.startsWith
8(9)	java.lang.String.lastIndexOf
9(8)	Path.<init>
10(6)	java.util.Hashtable.get

せて示す．従来手法の取得にあたっては内部クラスも考慮したソースコードコーパスを作成し，同一解析対象による計測を実施している．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 7 の結果を見ると，5 行目の呼び出しの次の呼び出し文は，**println** メソッド呼び出し文であるが，この文は両方の結果とともに 11 位以下であることが分かる．この処理は **println** メソッド呼び出しがなくとも動作するため，次の次の呼び出し文である **hashMoreElements** 呼び出し文について考える．この文は従来手法ではコード補完リストの 3 番目になるが，本手法では 1 番目である．

同様に，表 8 の結果をみると，従来手法では 4 番目に存在し，本手法では 1 番目に存在する．表 9 の結果では，従来手法が 2 番目に存在し，本手法では 1 番目に存在する．いずれも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．

4.3 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 4.1 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 1 つである CheckstylePlugin 5.5.0 [*4] 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 37,699 行である．

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

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集計した値

```
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
Fig. 11 Procedure of evaluation.

表 10 CheckstylePlugin を用いた実験結果（従来手法）
Table 10 Evaluation results of CheckstylePlugin (past method).

n	存在する数	存在しない数	内部呼び出し数
5	127(79.4%)	32	212
10	62(86.1%)	10	124
15	37(86.0%)	6	79
20	30(88.2%)	4	45
25	24(100.0%)	0	23

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 10** に示し，本手法で測定した結果を**表 11** 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．また，本手法によってランキングがどのように変化したかについて測定した．表 11 における「存在する数」を，従来手法より順位が向上した数，低下した数，同順位であった数に分類した．結果を**表 12** に示す．

本手法で，n が 5 のときにコード補完リストに存在するメソッドは 125 個存在し，存在しなかったメソッドは 34

表 11 CheckstylePlugin を用いた実験結果（本手法）
Table 11 Evaluation results of CheckstylePlugin (this method).

n	存在する数	存在しない数	内部呼び出し数
5	125(78.6%)	34	212
10	58(80.6%)	14	124
15	35(81.4%)	8	79
20	30(88.2%)	4	45
25	22(91.7%)	2	23

表 12 従来手法と本手法とで順位の比較
Table 12 The comaprison of the ranking between this method and past method.

n	本手法の方が上位	従来手法の方が上位	同順位
5	38	32	55
10	28	5	25
15	15	5	15
20	10	6	14
25	2	7	13

個であった．また，内部の呼び出し文のメソッドは 212 個であったことが分かる．そして，125 個の内訳として，本手法の方が順位が上であった数が 38 あり，低下した数が 32 あり，同順位だった数が 55 であった．

4.4 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 5 は if 文の直後にくる呼び出し文の候補一覧であるが，もし，if が記述されていなかった場合は表 6 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 7，表 8，表 9 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．

ただし，表 10 と表 11 を比較すると，n が 20 以外の場合は本手法のほうが「存在する数」が低下している．従来手法の場合，補完したい箇所より前のすべての呼び出し文をキーとしてソースコードコーパスの問合せに用いるが，本手法ではそうではないことが影響している．本手法では，if 文の then 節と else 節間での呼び出し文の関係を取得しない．そのため，ソースコードコーパスに保存する呼び出し間の関係の情報量が少なくなる．また，補完したい箇所が else 節の中の場合，then 節の呼び出し文をキーとして用いないためである．

本手法によっていくつかコード補完リストに現れなくなったものがあるが，コード補完リストに現れた場合に，従来手法と比べて順位が向上したかについて考える．表 12 をみると，n がいずれの場合も，半分弱ほど同順位のままである．順位が変動したものに着目すると，n が 20 までは本手法のほうが順位が向上した数の方が多い．必ずしも，順位が向上するわけではないが，n が 10, 15, 20 あたりでは，向上する数が多い．補完したい箇所より前に記述された呼び出し文の数が 10 から 20 の場合，制御構造を考慮したほうがコード補完リストを有効活用できると考える．

しかし，順位が低下した数もある程度存在することが分かる．特に n が 25 になると低下した数のほうが上回る結果となった．ある程度以上の呼び出し文の情報がある場合は制御構造を考慮しなくても大きな順位の向上は見込めないかもしれない．

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ことができない問題は発生する．

5. 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 [7] 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．

ソースコード中の任意の場所で二分割し，その前後のコード片をペアとしてデータベースへ記録し，再利用に利用する研究がある [11]．この研究では，ソースコード中のすべてのトークンの並びを記録するため，ソースコードコーパスに多くの容量が必要となる．そこで，本研究ではメソッド呼び出し文の前後関係だけを記録するソースコードコーパスを採用している．

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．

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 [8] 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．PARSEWeb [9] は，既存のコードサーチエンジンを利用し，関係するソースコードを取得し，提示するツールである．これらのツールは，あるクラスから別のクラスの情報を取得するときに，どのようなメソッド呼び出しの連鎖が必要かを提示する．一方，我々のツールはあるメソッド呼び出し群があった場合に，次にくるメソッド呼び出しとして，どのような文が適切かを予測するツールであり，使用用途が異なる．また，Michail らは，アソシエーション分析を利用して，API の再利用パターンを抽出している [5], [6]．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．

6. おわりに

本稿では，メソッド呼び出し文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あげられる．

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, *Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp.213–222, ACM (2009).

[2] Holmes, R. and Murphy, G.C.: Using structural context to recommend source code examples, *Proc. 27th International Conference on Software Engineering* (*ICSE '05*), pp.117–125, ACM Press (2005).

[3] Holmes, R., Walker, R. and Murphy, G.: Approximate Structural Context Matching: An Approach to Recommend Relevant Examples, *IEEE Trans. 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, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol.40, No.6, pp.48–61, ACM (2005).

[5] Michail, A.: Data mining library reuse patterns using generalized association rules, *Proc. 22nd International Conference on Software Engineering*, pp.167–176, ACM (2000).

[6] Michail, A.: Browsing and searching source code of applications written using a GUI framework, *Proc. 24th International Conference on*, pp.327–337, ACM (2002).

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

[8] Sahavechaphan, N. and Claypool, K.: XSnippet: Mining For sample code, *Proc. 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, Vol.41, No.10, pp.413–430, ACM (2006).

[9] Thummalapenta, S. and Xie, T.: Parseweb: A programmer assistant for reusing open source code on the web, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp.204–213, ACM (2007).

[10] 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).

[11] 山本哲男，吉田則裕，肥後芳樹：ソースコードコーパスを利用したシームレスなソースコード再利用手法，情報処理学会論文誌，Vol.53, No.2, pp.644–652 (2012).

山本 哲男 （正会員）

平成 9 年大阪大学基礎工学部情報工学科卒業．平成 14 年同大学大学院博士後期課程修了．同年科学技術振興事業団計算科学技術研究員．平成 16 年立命館大学情報理工学部情報システム学科講師．平成 20 年同学科准教授．平成 23 年日本大学工学部情報工学科准教授．博士（工学）．ソフトウェア開発支援環境の研究に従事．電子情報通信学会，IEEE-CS 各会員．