情報処理学会論文誌 Vol.54 No.2 903–911 (Feb. 2013)

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

山本 哲男[1,a)]

受付日 2012年5月13日, 採録日 2012年11月2日

概要：ソースコードを記述していく際，開発者は，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て開発を行う．そこで，本研究では，既存ソースコードのメソッド呼び出し文に着目し，メソッド呼び出し文を補完する手法について提案する．あらかじめ，あるメソッド呼び出し文の前後に存在するメソッド呼び出し文をソースコードコーパスとして記録しておく．その後，コーディング時にソースコード中のメソッド呼び出しを取得し，それらの情報から，よく使われるメソッド呼び出し文を開発者に提示する．さらに，本稿では，提案手法を Eclipse プラグインとして実装して行っ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

Tetsuo Yamamoto[1,a)]

Received: May 13, 2012, Accepted: November 2, 2012

Abstract: When developers are writing source code, they often reuse existing source code or use libraries to develop effectively. We focus on method invocation statements in existing source code. This paper shows an approach to provide method invocation statements to complete the source code that the developer is writing. Source code corpus is made automatically from existing source code in advance. The corpus stores what method invocation statements are after/before a method invocation statement. The developer is recommended appropriate method invocation statement from the corpus. We implemented the approach as Eclipse Plugin, and describe the approach is effective through evaluations.

Keywords: code completion, IDE, corpus

1. はじめに

ソースコードを記述していく際，開発者は，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て開発を行うことがある．たとえば，**図 1** は Java でファイルの入力処理を行う典型的なソースコードであり，FileInputStream，InputStreamReader，BufferedReader クラスを利用し，readLine メソッドを呼び出すことでファイルの各行を読み込む処理である．あらかじめ用意されているこれらのクラスを利用することで，効率的にファイル入力処理が記述できる．しかし，クラス名やメソッド名が分からないと，自分で細かな処理を書く必要や既存のクラスを調べる時間が必要になり，手間がかかることになる．開発しているソースコード内のクラスやメソッドの処理は理解していても，そのソースコードで利用しているライブラリやフレームワークのクラスやメソッドを理解しているとは限らず，クラスやメソッド一覧を表示して順番にドキュメントを見ていく必要が生じる．そして，一覧を用いる場合，効率は見ていく順番や数に依存し

1 日本大学
Nihon University, Koriyama, Fukushima 963–8642, Japan
a) tetsuo@cs.ce.nihon-u.ac.jp

```java
try {
    FileInputStream fi = new FileInputStream("a.txt");
    InputStreamReader is = new InputStreamReader(fi);
    BufferedReader br = new BufferedReader(is);
    String str;
    while ((str = br.readLine()) != null) {
        System.out.println(str);
    }
    br.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

図 1　典型的な Java ソースコード片
Fig. 1　Typpical Java source code snippet.

てしまう．

さらに，開発者が Web を利用して検索するクエリの 34.2%は API に関する検索であり，その API に関するクエリの 64.1%が実際の API 名と違う名前であることが報告されている [2]．

そこで，多くの統合開発環境では，作成中のソースコードに対してコード推薦の機能を提供し，調べる手間や入力の手間を省く機能が存在する．これらの機能を用いると，ソースコード中の変数やクラス名などに対して，それらのメソッド一覧を提示することができる．しかし，どのクラスを利用すべきか分からなければ，メソッドのコード補完を利用ができないという問題点がある．さらに，API に精通していない開発者の場合，すべての候補の API ドキュメントを確認し，利用可能か判断する必要が生じ，候補が多い場合は調べるのに時間がかかるという問題点もあげられる．

また，近年，世の中の人が作成・利用した情報を集めて解析し，推薦する仕組みやシステムが多く存在する．これらは，人々の集合知を利用することで，利用者の求めているものを推薦する仕組みである．我々はソースコードも同様な考えに基づいて推薦することで，推薦精度が上がるのではないかと考える．つまり，世の中の人に多く利用されているソースコードが品質の高いソースコードではないかと考え，その情報を利用する．

そこで，本研究では，既存ソースコードでよく使われるメソッド呼び出し文を開発者に提示することで，開発効率を上げることを目標にする．あらかじめ多くのソースコードから，あるメソッド呼び出し文の前後に存在するメソッド呼び出し文を記録しておく．

本研究の特徴は以下のとおりである．

- メソッドのみの補完をするのではなく，メソッド呼び出し文自体を補完対象とする．これにより，クラス名が分からなくても補完の候補を利用可能になる．
- すべての候補を提示させるのではなく，既存のソースコードでよく記述されている呼び出し文のならびを基に候補を作成する．これにより，可能性の高い候補から探すことでき，調べる時間が短縮できる．

本手法では，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を利用者として想定している．開発者が API の機能を開発で利用しようとする際には，API マニュアルをひととおり読む必要があるといえる．

さらに，本研究では，提案する仕組みを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，提案手法で推薦する呼び出し文が妥当であるかの評価も実施した．

以降，2 章で提案手法を説明し，3 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．4 章では，関連研究について触れ，最後に 5 章で本稿をまとめる．

2.　提案手法

本章では，我々が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．本手法はクライアント・サーバモデルをとっており，サーバで実現する手法とクライアントで行われる手法からなる．サーバでは，補完の元となる情報をデータベースに保存しており，補完要求をするクライアントの問合せに応答し，補完情報を返す．データベースの内容は既存のソースコードのメソッド呼び出し文（以降，呼び出し文とよぶ）を解析したものになる．具体的には呼び出し文の前後の呼び出し文を調べ保存する．本研究ではこのデータベースをソースコードコーパスとよぶ．クライアントは，現在入力中のソースコードを解析し，それらの情報をサーバに渡し補完情報を取得し，開発者に提示する．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．

本手法の利用状況として以下の状況を想定している．開発者がソースコードを記述する場合，メソッド単位に作成していく．メソッドを実装する際は先頭から処理を記述していく．ただし，開発環境などが自動的に補完・生成するソースコードなどは除く．このように，あるメソッド呼び出しに必要な処理は事前にメソッドの先頭に記述されていることを想定している．

2.1　補完手法の概要

本手法を利用したコード補完の流れを図 **2** にしめす．処理の流れはソースコードコーパス作成処理と，作成しているソースコードに対するコード補完処理に分けられる．本手法の最も重要な要素は，既存のソースコードの中から，あるメソッド呼び出し文の前後に存在するメソッド呼び出し文の割合を計算し，ソースコードを補完することである．

事前に既存のソースコードを解析し，その情報をデータベースにソースコードコーパスとして保存する処理が必要である．まず，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，文と文の組合せをソースコードコーパスへ保存する．たとえば，あるメソッド内に 5 つの呼び出し文が存在した場合，

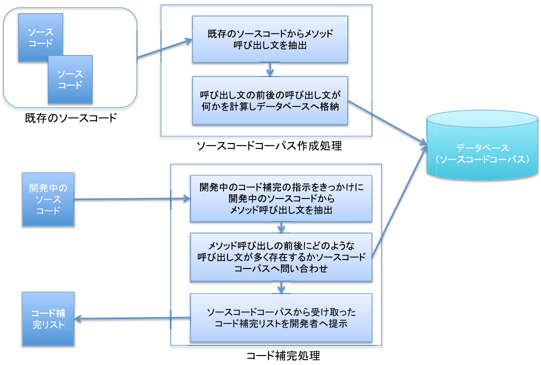

図 2　コード補完の流れ

Fig. 2　Flow of code completion.

10 通りの組合せが存在するため，そのすべての組合せをデータベースに保存する．

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データベース内に存在した場合は，出現数の値を 1 増やすだけとする．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 1 度作成すればよく，ソースコードを追加したい場合は，既存のデータベースへ解析した情報を保存するだけですむ．

そして，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．指定した場所から，その場所が含まれるソースコード中のメソッドを特定し，その場所の前後に存在する呼び出し文を取得する．取得はサーバでのソースコード解析と同様の方法で行う．

たとえば，図 1 のソースコードで 5 行目から 9 行目までを記述しておらず，まだ 1 行目から 4 行目と 10 行目から 12 行目までしかソースコード中に記述していないとする．そして，開発者が 4 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，クライアントでは，指定位置の前に存在する呼び出し文として，2 行目の FileInputeStream コンストラクタ呼び出し，3 行目の InputStreamReader コンストラクタ呼び出しと 4 行目の BufferdReader コンストラクタ呼び出しを抽出する．さらに，11 行目に存在する呼び出し文として，Exception の printStackTrace 呼び出しを抽出する．なお，オブジェクト生成の new 文はコンストラクタ呼び出しとして考え，呼び出し文の 1 つとして扱う．サーバでの解析処理でも同様に扱い処理する．その後，前に存在するメソッド呼び出し文のリスト（前リストとよぶ）と後に存在する呼び出し文のリスト（後リストとよぶ）をサーバに渡し，コード候補のための呼び出し文のリストを取得する．コードを補完したい場所によっては後リストが存在しない場合が存在する．その場合，後リストは空として扱い，前リストだけを用いる．

for all M in すべてのメソッド **do**
　メソッド呼び出し文のリスト L を空に初期化
　for all m in M 内のすべてのメソッド呼び出し文とインスタンス生成文 **do**
　　if m がメソッド呼び出し文 **then**
　　　m を「クラス名の完全限定名．メソッド名」に変換
　　else
　　　m を「クラス名の完全限定名．<init>」に変換
　　end if
　　L の最後に m を追加
　end for
　for all x in L **do**
　　for all y in L のサブリスト（x の次から最後まで） **do**
　　　// k 個の呼び出し文が存在した場合，${}_kC_2$ 個の組み合わせになる
　　　if 三つ組 A（x，y，任意の値 n）がソースコードコーパスに存在 **then**
　　　　ソースコードコーパス中の三つ組 A を（x，y，$n+1$）に更新
　　　else
　　　　ソースコードコーパスに（x，y，1）を追加
　　　end if
　　end for
　end for
end for

図 3　コーパス作成アルゴリズム

Fig. 3　Algorithm of corpus creation.

呼び出し文のリストを受け取ったサーバは，まず，前リストの各要素（呼び出し文）に対して，その呼び出し文の後に続く呼び出し文のリスト（結果リストとよぶ）をソースコードコーパスから取得する．取得するときに出現数も同時に取得しておく．さらに，後リストも同様に処理する．ただし，後に続く呼び出し文ではなく，前に存在する呼び出し文のリストを取得する．例の場合，前リストは 3 つの呼び出し文，後リストには 1 つの呼び出し文が存在するため，合計 4 つの結果リストをソースコードコーパスから取得する．これらの 4 つのリストを 1 つのリストに統合し，補完に適した順番に並べ替える．そして，このリストをクライアントにコード補完のリストとして返す．

クライアントは，受け取ったリストを用いて，開発者に既存のソースコードでは次にどのような呼び出し文がよく使われているかを提示する．

2.2 サーバでの処理

本節では，サーバで実行されるソースコードコーパス作成の処理とソースコードコーパスからコード補完リストを取得する処理について説明する．

2.2.1 コーパス作成処理

コーパス作成時のアルゴリズムを疑似コードを用いて図 3 に示す．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・意味解析が終了しており，ソースコード中のすべてのメソッドは抽出できているものとする．

メソッドの内容が図 1 である場合を例として，上記の流

れを説明する．メソッド中のすべてのメソッド呼び出し文とインスタンス生成文を抽出すると，図 4 の左側の 7 つの文となる．次に，各呼び出し文のクラス名を特定し，クラス名の完全限定名を取得する．その後，メソッド名を最後に足すと図 4 の右側の文となる．

これら 7 つの文の組合せは 21 個あり，それぞれをソースコードコーパスへ登録する．たとえば，1 つめと 2 つめの組合せは以下のとおりとなる．出現数はすでにソースコードコーパスにこの呼び出し文の組合せが存在するかどうかで変わる．

```
(java.io.FileInputStream.<init>,
 java.io.InputStreamReader.<init>, 1)
```

なお，呼び出し文の引数の情報は格納しない．そのため，引数が異なる同じメソッド名は同じ文として処理する．また，呼び出し文の引数にメソッド呼び出し文があるといった，字面上とプログラムの呼び出し処理の順番が異なる場合に，どちらを先にするかは構文解析の実装に依存するものとする．ただし，クライアントで利用する構文解析の実装も同等の処理に合わせる．これらの処理をすべてのメソッドについて行うことで，ソースコードコーパスを作成していく．

2.2.2　コーパス問合せ処理

ソースコードコーパスからコード補完のための候補を取得する方法について説明する．処理の入力は 2 つあり，それぞれ，前リストと後リストとよぶ．それぞれのリストは呼び出し文のリストである．そして，出力はランク付けされた呼び出し文のリストになり，コード補完リストとよぶ．

疑似コードを用いたアルゴリズムを**図 5** に示す．前リストに含まれる呼び出し文の後に続く呼び出し文をソースコードコーパスから取得し，その割合を計算する．後リストについても同様のアルゴリズムで処理を行う．ただし，疑似コード中のコメント 1 は実行せず，コメント 2 の処理で L の 1 番目の要素となっている箇所を 2 番目の要素としてソースコードコーパスから取得する．さらに，コメント 3 でペアを生成する際に，2 番目の要素を用いて作成する部分を 1 番目の要素を用いる．作成したコード補完リストを結果として出力する．リストの各要素は（呼び出し文，値）のペアであり，値でソートされている．

表 1 は Eclipse ソースコードの Java ファイルをすべてソースコードコーパスへ登録したときに，java.io.PrintStream の println メソッドの後にくるメソッドの上位 5 つ（出現数順）を表している．リスト中に三つ組は 7,173 個あり，出現数の合計は 392,795 であった．この値から，1 番目の java.io.PrintStream.println の割合は 0.446 となる．

```
new FileInputStream("a.txt")        java.io.FileInputStream.<init>
new InputStreamReader(fi)           java.io.InputStreamReader.<init>
new BufferedReader(is)              java.io.BufferedReader.<init>
br.readLine()               ⇒       java.io.BufferedReader.readLine
System.out.println(str)             java.io.PrintStream.println
br.close()                          java.io.BufferedReader.close
e.printStackTrace()                 java.lang.Exception.printStackTrace
```

図 4　メソッド呼び出し文の抽出
Fig. 4　Extracting method invocation statements.

コード補完リスト C を空に初期化 // 1
for all x in 前リスト **do**
　出現数の合計 $SUM \Leftarrow 0$
　結果リスト R を空に初期化
　for all L in ソースコードコーパスのすべての三つ組 **do**
　　if L の 1 番目の要素 $==$ x **then**
　　　// 2
　　　$SUM \Leftarrow SUM + L$ の 3 番目の要素
　　　R にペア（L の 2 番目の要素，L の 3 番目の要素）を追加
　　　// 3
　　end if
　end for
　// 出現数から出現割合に変更
　for all r in 結果リスト R **do**
　　r を（r の 1 番目の要素，r の 2 番目の要素 $/SUM$）に変換し R を更新
　end for
　// ここから結果リストを補完リストへ統合していく処理
　for all r in 結果リスト R **do**
　　for all c in コード補完リスト C **do**
　　　if r の 1 番目の要素 $==$ c の 1 番目の要素 **then**
　　　　c を（c の 1 番目の要素，r の 2 番目の要素 $+$ c の 2 番目の要素）を変換し，C を更新（☆）
　　　end if
　　end for
　　if r を用いて C を更新していなければ（☆が実行されていなければ） **then**
　　　ペア r を C に追加
　　end if
　end for
end for
コード補完リスト C を 2 番目の要素でソート

図 5　コーパス問合せアルゴリズム
Fig. 5　Algorithm of corpus query.

表 1　java.io.PrintStream の println メソッドの後につづくメソッド一覧
Table 1　Methods after println method of java.io.PrintStream class.

出現数	呼び出し文
175,498	java.io.PrintStream.println
19,776	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpression
12,096	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpressionWithName
5,944	org.eclipse.jdt.internal.compiler.parser.Parser.consumeUnaryExpression
5,916	org.eclipse.jdt.internal.compiler.parser.Parser.consumeAssignmentOperator

2.3 クライアントでの処理

クライアントでは開発者が作成中のソースコードから必要な呼び出し文の情報を抽出し，その情報をもとにソースコードコーパスからコード補完リストを取得して提示する．開発者は，利用中のエディタや統合開発環境などから編集中のソースコードとそのソースコードの補完したい箇所を明示する．その後の処理の流れを以下に示す．

(1) 明示したソースコードを構文解析・意味解析する．
(2) 明示した箇所を特定し，メソッドの中でなければ何もせずに終了する．メソッドの中であれば，明示した箇所の前に存在するメソッド呼び出し文と後に存在する呼び出し文をサーバで行っている処理と同様の変換方法で変換し，それぞれを前リスト，後リストとする．
(3) 2 つのリストをサーバへ渡しコード補完リストを問い合わせる．
(4) サーバから入手したコード補完リストを開発者へ提示する

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．

2.4 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．

ソースコードコーパスに三つ組を登録するときに，三つ組の 1 番目の要素である前の呼び出し文と 2 番目の呼び出し文である後のメソッド呼び出し文をともにキーにすることで，検索時の処理の高速化をする必要がある．また，出現数が小さく上位にはほとんど出ることのない三つ組が数多く存在することになるため，出現数にソートして保存しておき，リストを取得する際に閾値を利用して，上位しか取得しないことで高速化を図る．

クライアントは Eclipse プラグインとして実装した．開発者が Eclipse の Java エディタ上でソースコードを記述している際に，開発者のトリガによって書きかけのソースコードの解析が始まる．解析後に得られた呼び出し文のリストを用いてソースコードコーパスへ問い合わせ，コード補完リストを取得する．リストを取得する際に，サーバではリストの結合処理が必要になるが，出現割合の足し込み処理だけになるので，高速に処理が終わる．また，各結果リストの取得もキーとなる呼び出し文を用いて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，高速に処理が可能である．

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

3. 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．

3.1 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利用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 Eclise のソースコードを用いた．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．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 i7 2.8 GHz でメモリが 16 GB の計算機を利用したところ，構築時間は 1 時間 2 分 40 秒かかり，データベースのサイズは 4.2 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 1 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．

クライアントからソースコードコーパスへの問合せ時のパフォーマンスは図 1 のソースコードを利用し，各呼び出し文の後でトリガをかけて計測した平均値は 650 ms 程度であった．ただし，この時間には起動時に 1 度実行するデータベースの初期化処理と Eclipse でコード補完リストを表示する処理は含まれていない．クラインアントのソースコードの解析時間とソースコードコーパスからのリスト取得とリストの統合時間の和が計測時間となる．この時間から，1 度起動してしまえばほとんど問題にならない時間と考える．

3.2 適用例

3.1 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 のソースコードで 5 行目から 9 行目までを記述しておらず，まだ 1 行目から 4 行目と 10 行目から 12 行目までしかソースコード中に記述していないとする．そして，開発者が 5 行目でコード補完をしたいと思った場合を例とする．

コード補完する際にクライアントでは，前リストと後リストを作成する．前リストは，2 行目から 4 行目までの 3

表 2　java.io.FileInputStream<init>の後にくる文一覧
Table 2　MethodCalls after java.io.FileInputStream<init>.

順位（割合）	呼び出し文
1（3%）	java.io.InputStream.close
2（2%）	java.io.FileInputStream.close
3（1%）	java.io.PrintStream.println
4（1%）	java.util.Properties.load
5（1%）	java.io.File.<init>
6（1%）	NLS.bind
7（1%）	Status.<init>
8（1%）	java.io.FileInputStream.<init>
9（1%）	java.io.FileOutputStream.<init>
10（1%）	java.io.IOException.<init>

表 3　java.io.InputStreamReader<init>の後にくる文一覧
Table 3　MethodCalls after java.io.InputStreamReader<init>.

順位（割合）	呼び出し文
1（4%）	java.lang.StringBuffer.append
2（4%）	java.io.BufferedReader.readLine
2（4%）	java.lang.String.equals
3（3%）	java.util.Map.get
4（2%）	java.lang.StringBuffer.toString
5（2%）	java.io.BufferedReader.close
6（2%）	java.lang.String.substring
7（2%）	java.lang.String.startsWith
8（2%）	java.lang.StringBuffer.<init>
9（1%）	org.eclipse.jdt.internal.compiler.impl.CompilerOptions.updateSeverity
10（1%）	java.util.Map.put

表 4　java.io.BufferedReader<init>の後にくる文一覧
Table 4　MethodCalls after java.io.BufferedReader<init>.

順位（割合）	呼び出し文
1（6%）	java.io.BufferedReader.readLine
2（6%）	java.io.InputStreamReader.<init>
3（3%）	java.io.BufferedReader.close
4（3%）	java.lang.StringBuffer.append
5（2%）	java.lang.String.substring
6（2%）	java.lang.String.startsWith
7（2%）	java.lang.String.trim
8（2%）	java.lang.String.indexOf
9（1%）	java.util.StringTokenizer.nextToken
10（1%）	java.io.PrintStream.println

つのインスタンス生成文になり，後リストは 11 行目の呼び出し文になる．それぞれの文をソースコードコーパスに問い合わせると，**表 2**，**表 3**，**表 4**，**表 5** 示す結果が得られる．ただし，11 位以下もしくは 1%未満の文は省略している．これらのリストを統合した結果，コード補完リストとして開発者に提示されるリストを**表 6** に示す．

java.io.BufferedReadline.readLine がコード補完リストの 1 位となり，例に使用したソースコードの次にくる文と一致することが分かる．

表 5　java.lang.ExceptionprintStackTrace の前にくる文一覧
Table 5　MethodCalls before java.lang.Exception.printStackTrace.

順位（割合）	呼び出し文
1（2%）	java.io.PrintStream.println
2（1%）	java.lang.Exception.printStackTrace

表 6　図 1 の 5 行目でのコード補完リスト
Table 6　Code completion list at line 5 in Fig. 1.

順位（値）	呼び出し文
1（10）	java.io.BufferedReader.readLine
2（7）	java.lang.StringBuffer.append
3（7）	java.io.InputStreamReader.<init>
4（5）	java.lang.String.equals
5（5）	java.io.BufferedReader.close
6（5）	java.io.PrintStream.println
7（4）	java.lang.String.substring
8（4）	java.lang.String.startsWith
9（4）	java.io.InputStream.close
10（3）	java.lang.StringBuffer.<init>

```
1  try {
2      ZipFile sourceZipFile = new ZipFile("Example.
           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 6　ZIP ファイルを処理する Java ソースコード片
Fig. 6　Java source code snippet to search a file in ZIP file.

同様な測定を**図 6** に示すソースコードに対しても行った．**表 7**，**表 8**，**表 9**，**表 10** はそれぞれ図 6 で 1 行目から 5，8，9，20 行目までしかソースコードを記述していない場合，その直後でトリガをかけた場合のコード補完リストの結果を示す．

表 7 の結果を見ると，5 行目の次には，println メソッド呼び出し文がくるが，ロジックの文ではないため無視をしたとすると，9 行目の hashMoreElements 呼び出し文がくる．この文はコード補完リストの 3 番目になる．同様に，次に来るメソッドがコード補完リストの何番目にくるか見

表 7 図 6 の 5 行目直後でのコード補完リスト
Table 7 Code completion list after line 5 in Fig. 6.

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

表 8 図 6 の 8 行目直後でのコード補完リスト
Table 8 Code completion list after line 8 in Fig. 6.

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

表 9 図 6 の 9 行目直後でのコード補完リスト
Table 9 Code completion list after line 9 in Fig. 6.

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

ると，表 8 の結果は 4 番目に，表 9 の結果は 2 番目に，表 10 の結果は 8 番目にそれぞれ存在する．

ただし，11 行目の getName() の直後で補完リストを取得した場合，toLowerCase() は補完リストには存在せず（1%未満の三つ組を取得時に無視した場合），toLowerCase() 直後で補完リストを取得した場合，indexOf() は 52 番目に存在した．

表 10 図 6 の 20 行目直後でのコード補完リスト
Table 10 Code completion list after line 20 in Fig. 6.

順位（値）	呼び出し文
1（140）	java.io.PrintStream.println
2（23）	java.util.Enumeration.nextElement
3（21）	java.lang.String.substring
4（19）	java.lang.String.length
5（16）	java.lang.StringBuffer.append
6（15）	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpression
7（14）	java.util.zip.ZipEntry.getName
8（13）	java.util.zip.ZipFile.close
9（12）	java.lang.String.equals
10（11）	java.util.Enumeration.hasMoreElements

```
if メソッド中の呼び出し文の数 >= n + 1 then
  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
    「内部呼び出し数」を 1 つ増やす
  else
    1 番目から n 番目のすべての呼び出し文を前リストとしてソー
    スコードコーパスへ問合せコード補完リストを取得（後リス
    トは作成しない）
    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
      「候補リストに存在する数」を 1 つ増やす
    else
      「候補リストに存在しない数」を 1 つ増やす
    end if
  end if
end if
```

図 7 計測手順
Fig. 7 Procedure of evaluation.

3.3 有効性の評価

既存のソースコードを利用して，候補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 3.1 節で作成したものを用い，候補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 *3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．

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，「候補リストに存在する数」,「候補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 **7** 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 7 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．

*3 http://eclipse-cs.sourceforge.net/

n を 5，10，15，20，25 として測定した結果を**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，候補リストに存在する数の中で，何番目に存在するかも判定し，計測した結果を**表 12** 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．

n が 5 のとき，候補リストに存在するメソッドは 117 個存在し，存在しなかったメソッドは 33 個であった．また，6 番目の呼び出し文が内部の呼び出し文のメソッドは 201 個であったことが分かる．つまり，118 個の Java ファイル中に $n+1$ 個（6 個）以上のメソッドは，351 個あったことになる．そして，117 個の内訳として，上位 5 位以内に 6 番目の呼び出し文が含まれているメソッドは 65 個あり，10 位以内は 76 個あったことが分かる．

3.4 考察

ファイルの読み書きや ZIP ファイルの処理といった特定の処理のための API では，補完リストの上位に期待する呼び出し文がくることが分かる．このような典型的な構文に対しては補完リストが有効ではないかと考える．しかし，String クラスといった，どの場面でも使うようなライブラリは，ソースコードコーパスに登録したソースコードに依存してしまう．

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．クラス名が分かっておりメソッド名だけが欲しい場合には，候補リストから，クラス名でフィルタリングを行い提示すればさらに効率が上がると考える．既存の統合開発環境でも実現しているメソッド名補完機能と似ているが，候補に出すメソッドをしぼり，よく使われる順に並べることができる．たとえば，表 7 で Enumeration クラスにしぼれば，1 番目が適切なメソッド呼び出し文になる．同様に考えてリストを見ると，表 8 では 1 番目に，表 9 では 1 番目に，表 10 でも 1 番目となる．クラス名が分かれば，さらに精度が向上することが分かる．

実際の Eclipse Plugin のソースコードをもとにした実験では，呼び出し文を記述する際に，半分近くが内部のメソッド呼び出し文であることが分かる．開発者が作成しているソフトウェアのクラスやメソッドの情報は理解しているものとし，どのクラスを利用すればよいかは分かることが多く，メソッド名のコード補完が重要になってくる．本手法はソースコードコーパスにソフトウェアの呼び出し関係を登録することで，補完を実現しており，開発中のソフトウェアを登録していない場合は，コード補完リストに内部呼び出し文が含まれない．

残りの外部のクラスの呼び出し文が候補リストに含まれるのかどうか，含まれる場合は上位に適切な呼び出し文がくるのかについて考える．表 11 を見ると，8 割が候補に含まれることが分かり，さらに，表 12 を見ると，その中の上位 5 番以内に出現する割合は 5 割以上ある．20 番以内の場合，7 割から 8 割がコード補完リストに含まれる．開発者がコード補完リストの上位 20 件を見るだけで，8 割程度の確率で適切な呼び出し文が選べる．ただし，コード補完リストに現れない場合も 2 割あり，その場合は，API ドキュメントやサンプルソースコードを参照するといった，従来の調べ方で調べる必要がある．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ため効果的な結果が出ている．

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ため，候補リストに存在しない数として計測される．Eclipse のプラグインを用いた評価では，コーパスに Eclipse の本体だけを格納しており，それ以外のソフトウェアは解析していないが，同じドメインのソフトウェアの場合，2 割程度におさえることができる．

表 11 CheckstylePlugin を用いた実験結果
Table 11 Evaluation results of CheckstylePlugin.

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 12 候補リストの上位に含まれる数
Table 12 The number of methodcall in high ranking of code completion list.

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%）

4. 関連研究

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使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 [5] 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 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 [6], [7]．Strathcona [3], [4] 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．

ソースコード中の任意の場所で 2 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 [10]．この研究では，既存のソースコードを大量にデータベースへ登録することで，よく利用されるコード片を見つけることが可能となる．同様に，本研究でも既存のソースコードをデータベースへ蓄積しておき，その情報を補完に利用する．ただし，ソースコードコーパスとして作成するトークンの種類や粒度が異なる．また，コード全体を補完するのか，メソッド呼び出し文のみに着目して補完するかの違いがある．文献 [10] では，ソースコード中のすべてのトークンのならびを記録するため，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．

5. おわりに

本稿では，既存ソースコードのメソッド呼び出し文に着目し，メソッド呼び出し文を補完する手法について提案した．あらかじめ，あるメソッド呼び出し文の前後に存在するメソッド呼び出し文をソースコードコーパスとして記録しておき，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て行った評価実験を行い，候補の上位に適切なメソッド呼び出し文がくることを確認した．今後は，様々な種類のソースコードに対してソースコードコーパスを作成し，実験することがあげられる．また，メソッド呼び出し文だけでなく，そのメソッド呼び出し文のエラーチェックルーチンといった，関係のあるソースコードも提示することで，さらなる開発の効率化をめざすこともあげられる．

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*, New York, USA, pp.213–222 (2009).

[2] Hoffmann, R., Fogarty, J. and Weld, D.: Assieme: Finding and leveraging implicit references in a web search interface for programmers, *Proc. 20th Annual ACM Symposium on User Interface Software and Technology*, pp.13–22 (2007).

[3] Holmes, R. and Murphy, G.C.: Using structural context to recommend source code examples, *Proc. 27th International Conference on Software Engineering*, New York, USA, pp.117–125 (2005).

[4] Holmes, R., Walker, R. and Murphy, G.: Approximate Structural Context Matching: An Approach to Recommend Relevant Examples, *IEEE Trans. Softw. Eng.*, Vol.32, No.12, pp.952–970 (2006).

[5] 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 (2005).

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

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

[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 (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 (2007).

[10] 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．