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)]

概要：ソースコードを記述していく際，開発者は，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，コーディング時の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．

A Code Completion of Method Invocation Statement using Source Code Corpus

Tetsuo Yamamoto[1,a)]

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.

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て開発を行うことがある．例えば，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．

[1] 日本大学
Nihon University
[a)] tetsuo@cs.ce.nihon-u.ac.jp

さらに，開発者が Web を利用して検索するクエリーの 34.2%は API に関する検索であり，その API に関するクエリーの 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，メソッドのコード補完では，クラス内だけの情報しか扱わない，候補が多い場合は分かりにくいといった問題点が存在する．さらに，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は利用ができない．

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

© 2012 Information Processing Society of Japan

```
1  try {
2      FileInputStream fi = new FileInputStream("a.txt
           ");
3      InputStreamReader is = new InputStreamReader(fi
           );
4      BufferedReader br = new BufferedReader(is);
5      String str;
6      while ((str = br.readLine()) != null) {
7          System.out.println(str);
8      }
9      br.close();
10 } catch (Exception e) {
11     e.printStackTrace();
12 }
```

図 1　典型的な Java ソースコード片

Fig. 1　Typpical Java Source Code Snippet

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もと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．ある開発者と同じドメインのソフトウェアを開発しているオープンソースの開発者が作成したソースコードへ集合知の考えを適用することで，有能な開発者が作成した「知」を収集・利用でき，開発の生産性向上につながると考える．

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，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あらかじめ多くのソースコードを用いて記録しておき，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．本研究の特徴は以下のとおりである．

- 既存のソースコードを用意するだけで利用可能である．
- 必要になりそうな候補のみを表示させ，実際に記述するのは開発者とすることで，開発者が理解できないようなソースコードの生成はしない．
- 提案手法を Eclipse プラグインとして実装することで，開発中に推薦機能を容易に利用可能である．

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

本節では，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い合わせに応答し，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，

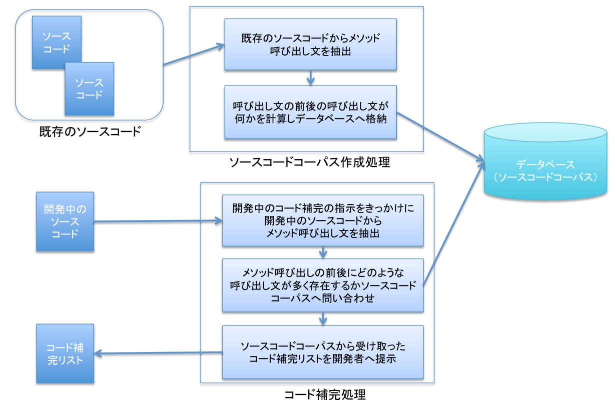

図 2　コード補完の流れ

Fig. 2　Flow of Code Completion

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．

以降，最初に全体の流れの説明をし，次に，サーバで利用する手法，クライアントで利用する手法の順に説明する．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．

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，ソースコード補完をする事である．

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，文と文の組み合わせをソースコードコーパスへ保存する．例えば，あるメソッド内に 5 つの呼び出し文が存在した場合，10 通りの組み合わせが存在するので，その組み合わ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み合わ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

© 2012 Information Processing Society of Japan

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．

例えば，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．この時，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，呼び出し文の一つとして扱う．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．

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ので，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とになる．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び替える．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

コーパス作成時の処理の流れを以下に示す．

(1) ソースコードを構文解析，意味解析し，ソースコード中のクラス内のメソッド一覧を取得する．

(2) 各メソッド中のメソッド呼び出し文とインスタンス生成文を取得する．

```
new FileInputStream("a.txt")
new InputStreamReader(fi)
new BufferedReader(is)
br.readLine()
System.out.println(str)
br.close()
e.printStackTrace()
```

```
java.io.FileInputStream.<init>
java.io.InputStreamReader.<init>
java.io.BufferedReader.<init>
java.io.BufferedReader.readLine
java.io.PrintStream.println
java.io.BufferedReader.close
java.lang.Exception.printStackTrace
```

図 3　メソッド呼び出し文の抽出

Fig. 3　Extracting Method Invocation Statements

(3) 呼び出し文は，すべて「クラス名の完全限定名.メソッド名」に変換する．インスタンス生成文の場合，「クラス名の完全限定名.<init>」という名前に変換する．

(4) 呼び出し文とインスタンス生成文の組み合わせを作成する．メソッド中に n 個の呼び出し文が存在した場合，${}_nC_2$ 個の組み合わせになる．

(5) 組み合わせをソースコードコーパスへ登録する．登録する情報は（前の呼び出し文，後の呼び出し文，出現数）の三つ組である．もし，ソースコードコーパス内に（前の呼び出し文，後の呼び出し文，出現数は任意）の三つ組がすでに存在していれば，出現数を 1 増やした三つ組に更新する．なければ，出現数を 1 にして新規に登録する．

ソースファイルを解析し，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 3 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 3 の右側の文となる．

これら 7 つの文の組み合わせは，21 個あり，それぞれをソースコードコーパスへ登録する．例えば，1 つめと 2 つめの組み合わせは以下の通りとなる．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み合わ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 コーパス問い合わせ処理

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呼ぶ．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る．

© 2012 Information Processing Society of Japan

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
175498	java.io.PrintStream.println
19776	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpression
12096	org.eclipse.jdt.internal.compiler.parser.Parser.consumeBinaryExpressionWithName
5944	org.eclipse.jdt.internal.compiler.parser.Parser.consumeUnaryExpression
5916	org.eclipse.jdt.internal.compiler.parser.Parser.consumeAssignmentOperator

最初に，前リストに含まれるすべての呼び出し文について以下の処理を繰り返す．その結果生成されるリストを結果リストと呼び，前リストの各呼び出し文ごとに作成される．

(1) リストに含まれる呼び出し文と同じ文を三つ組の 1 番目の要素にもつ三つ組みをソースコードコーパスからすべて取得する．
(2) 取得した三つ組の 3 番目の要素である出現数を合計する．
(3) 三つ組の 2 番目の要素である呼び出し文と 3 番目の要素である出現数を（2）で計算した合計数で割った値を要素とするペアを新たに作成する．つまり，ペアは（呼び出し文，割合）となる．
(4) 結果リストに（3）のペアを追加する．

同様に，後リストに含まれるすべての呼び出し文についても同様の処理を繰り返す．ただし，（1）の処理で 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，（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．

次に，それぞれの結果リストを統合し，一つのコード補完リストにする処理について説明する．まず，全ての結果リストを繋げた 1 つのリストを作成する．その後，ペアの 1 番目の要素である呼び出し文が同じものがリスト中に存在する場合は，割合を数値とみなし，2 つのペアの値を足した新しいペアを作成し，統合する前に 2 つのペアは消去する．これを 1 番目の要素がリスト中に 2 つ以上存在しなくなるまで繰り返す．最後に，2 番目の要素の値でリストをソートする．

このソートしたリストをコード補完リストとして出力する．リストの各要素は（呼び出し文，値）のペアとなる．

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 7173 個あり，出現数の合計は 392795 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．

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等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みを登録する時に，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共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

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

© 2012 Information Processing Society of Japan

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，開発者のトリガーによって書きかけのソースコードの解析がはじまる．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．

3. 評価

提案手法を評価するために実験を行った．本節では，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 2.8GHZ でメモリが 16GB 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 4.2GB 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．

クライアントからソースコードコーパスへの問い合わ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ーをかけて計測した平均値は 650ms 程度であった．ただし，この時間には起動時に一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，一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 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 に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．コード補完リストの 1 位には java.io.BufferedReadline.readLine 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．

同様な測定を図 4 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 4 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，その直後でトリガー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

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

© 2012 Information Processing Society of Japan

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．

3.3 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

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

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 Figure 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 4 ZIP ファイルを処理する Java ソースコード片

Fig. 4 Java Source Code Snippet to Search a File in ZIP File

表 7 図 4 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 Figure 4

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 4 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 Figure 4

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

© 2012 Information Processing Society of Japan

りは，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．例えば，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．

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，JDK 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．しかし，ある特定のライブラリの処理といった補完の場合は，そのライブラリを利用しているソースコードを数多くソースコードコーパスに登録する必要がでてくる．少ない場合にどのような結果が出るのか評価する必要があると考える．

表 9 図 4 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 Figure 4

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

表 10 図 4 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 Figure 4

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

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．

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[6] や Xsnippet[10] 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[11] 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 [7], [8], [9]．Strathcona[3], [4], [5] 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．

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，実験することが上げられる．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挙げられる．

© 2012 Information Processing Society of Japan

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, USA, ACM, 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, *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM, pp. 13–22, (2007).

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

[4] 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).

[5] Holmes, R., Walker, R. J. and Murphy, G. C.: Strathcona example recommendation tool, *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, Vol. 30, No. 5, pp. 237–240, (2005).

[6] 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).

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

[8] Michail, A.: CodeWeb : Data Mining Library Reuse Patterns, *Proceedings of the 23rd International Conference on Software Engineering*, pp. 827–828 (2001).

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

[10] 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).

[11] Thummalapenta, S. and Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web, *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, pp. 204–213, (2007).

© 2012 Information Processing Society of Japan