既存のコード資産を利用した制御構文補完機構

北原 元気[1,a)]　中野 圭介[1,b)]

概要：コード補完機構は，多くの統合開発環境 (以下，IDE) のエディタで提供されており，コーディングの利便性において重要な役割を果たしている． Eclipse や IntelliJ IDEA などの主要な IDE では，変数やメソッド呼び出しなどの単純な式だけでなく， if 文や for 文のような制御構文も補完することができる．しかし，これらの既存の補完機構では，補完されるコード内で使用される変数の選択が適切でないことや，ブロック内のコードの補完ができないといった問題点が存在する．本研究はこれらの問題を解決するために，既存のコード資産を利用した新しい制御構文補完機構を提案する．本機構では，コード検索とテストの実行の 2 つの過程を経て，補完候補を提示する．提示された候補は，適切な変数が利用され且つブロック内のコードも含まれているので，プログラマは望みの候補を選択することで，スムーズにコーディングを行うことができる. 本機構は， Java を対象に実装を行っている．

キーワード：コード補完, Java, IDE

1. はじめに

コード補完機構は，多くの IDE のエディタで提供されており，コーディングの利便性において重要な役割を果たしている．コード補完とは，不完全なコードに対して，次に書くことのできるコードの候補一覧を表示，挿入し，プログラマの入力の手間を軽減するものであり，多くのプログラマがこの機構を利用している [1]. IDE が持つコード補完機構は，主に変数やメソッド呼び出しのような単純な式を入力する際によく利用されている．長い名前の変数やメソッドを利用している場合，この機構によってプログラマのタイプ数を大きく減らすことが可能となる． Java では標準ライブラリでも長い名前のクラス名やメソッド名を採用しているので，コード補完機構はより有用性が高いと言える．

今まで，コード補完機構をより便利にしたツールの開発や研究が行われてきた [2]．たとえば，Code Recomenders プラグイン [3] は，メソッド呼び出しの補完において，過去の統計データをもとに，使用される可能性の高いメソッドを候補の上位に表示する．

一方で， IDE では変数やメソッド呼び出し以外にも様々なコードを補完することができる．その一例として `if` 文や `for` 文のような制御構文がある．これは，制御構文に必要な括弧や中括弧などを補完し，構文の大枠を自動生成するものである．特に Eclipse では，制御構文を補完する際の初期化式や条件式なども補完することができる．また，それらの式で利用される変数も，スコープ内から自動的に選択する．

しかし，制御構文の補完においては，内部で使用される変数が不適切なことや，内容の一部が異

[1] 電気通信大学大学院情報理工学研究科
[a)] kitahara@ipl.cs.uec.ac.jp
[b)] ksk@cs.uec.ac.jp

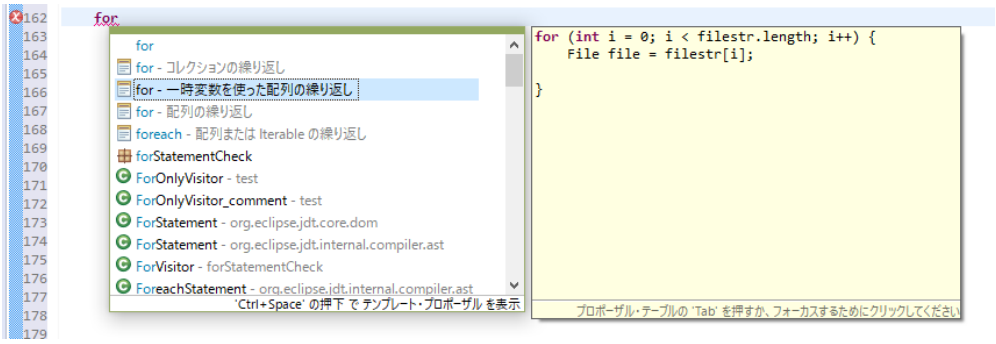

図 1　Eclipse での制御構文の補完の様子

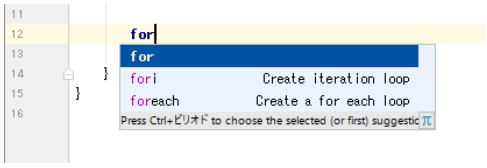

図 2　IntelliJ IDEA での制御構文の補完の様子

なっていることが多いため，補完候補を挿入した後に適宜修正する手間も生じてしまうことがある．実際に，Eclipse の制御構文の補完でもこのような問題が発生し，更にはブロック内のコードが補完できない．

本研究はこれらの問題を解決するために，既存のコード資産を利用した補完機構を提案する．補完候補は，既存のコード資産からの検索と，事前条件，事後条件をもとにしたテストの実行による絞り込みを行い予測する．また，本システムは Java を対象とし，Eclipse プラグインとして実装する．

本稿の構成は以下の通りである．2 節では簡単な例を用いてコード補完機構と補完候補の見つけ方について述べる．次に 3 節で提案機構の概要と設計について，4 節で本機構を構成するプログラムの実装について説明する．最後に 6 節で関連した研究やサービスについて述べ，7 節で今後の課題と本稿のまとめを述べる．

2.　制御構文に対するコード補完機構

コード補完機構は次の 3 つの動作から成り立つ．まず，プログラマからの不完全な入力を読み取る．不完全な入力とは，途中まで書かれた変数名や，メソッドチェーンなどである．次に，読み取った不完全なコードに対して，次に続くことのできるコードを検索する．最後に，提示した候補をある基準に従ってソートし，プログラマに候補の一覧を提示する．

本節では主要な IDE について，それぞれにおける制御構文の補完方法とその効果を for 文を例に説明し，問題点を述べる．

2.1　各 IDE での機構

Eclipse [4] や NetBeans [5] では，プログラマがエディタ上で for と記述した後にコード補完を行うコマンドを入力すると，図 1 のように補完候補がその概要を表す説明文とともにポップアップウィンドウに列挙され，実際に補完されるコードは別のポップアップウィンドウに表示される．プログラマはこれらの内容を見て，選択する補完候補を決定する．候補を選択すると，右ウィンドウのコードが挿入される．その後，プログラマは使用する変数を適宜変更し，ブロック内の処理を記述する．一部の補完候補にはブロック内のコードも記述されているが，そのコードの内容はローカル変数への配列要素の代入といった程度だけであり，それ以外の処理はやはりプログラマが記述する必要がある．

IntelliJ IDEA [6] でも，プログラマがエディタ上で for と記述すると，図 2 のように for 文に対する補完候補がその概要を表す説明文とともにポップアップウィンドウに列挙される．しかし，実際

```
1 boolean cmpSum(int[] first, int[] second){
2   int sum1 = 0, sum2 = 0;
3   for
4
5
6 }
```

(a) for 文の補完前の状況

```
1 boolean cmpSum(int[] first, int[] second){
2   int sum11 = 0, sum12 = 0;
3   for (int i = 0; i < second.length; i++){
4
5   }
6 }
```

(b) for 文の補完後の状況

図 3 for 文の補完例

に補完されるコードは見ることはできない．プログラマは説明文をもとに，選択する補完候補を決定する．また，あらかじめ用意されている候補の種類も，他の IDE よりも少ない．補完される for 文には初期化式や条件式の記述はなく，括弧や中括弧のみが補完されるものがほとんどである．

このように，主要の IDE には制御構文に対して入力を補助する機能がそれぞれ備わっている．

2.2 問題点

補完される制御構文内で使われる変数やブロック内の処理について，例を用いて考える．Eclipse において，2 つの配列の総和の大小を比較をする関数 `cmpSum(int[] first, int[] second)` というメソッドの作成中，プログラマはローカル変数 `sum1` の値を，`first` の各要素の総和にしようと考えていたとする．そこで， `int sum1 = 0, sum2 = 0;` と記述した後に `for` を記述し (図 3(a))，この際, 配列の要素数分だけループするような `for` 文を補完すると，図 3(b) の赤字で示されたコードが補完される．

ここで， `for` 文の継続条件式に使われる配列は，スコープ内で利用できるものがあれば，Eclipse は自動的にそれを利用する．利用できる配列が 2 つ以上存在する場合には，Eclipse は利用する配列を自動的に選択するが，その選択が適切とは限らないという問題点がである．このため，プログラマが意図した配列が選択されない可能性がある．この例の場合，プログラマは `first` の総和を求めたいので，継続条件式の `second.length` は適切ではない．そのため， `second` を `first` に書き換えるコストが生じてしまう．

また，Eclipse にあらかじめ用意されたコード補完は， `for` 文のブロック内を補完するように設計されていないという点も問題である．例のような “総和を求める” という多くの場面で利用されるであろう単純な処理であっても，プログラマはその処理内容を全て記述しなければならない．

3. 提案機構

効率的なソフトウェア開発を実現するために，コードの再利用という手法が用いられている．コードを再利用するための一般的な方法として，ライブラリの使用やコピー & ペーストなどが挙げられる．特にコピー & ペーストをすることで，複数行にわたるコード片も一瞬にして再利用することができる．ブロック内のコードも記述した候補を提示するために，本機構ではコードの再利用に基づいた方法で候補を探索する．

本節では，プログラマが Java で制御構文を記述するときに，プログラマの望む制御構文を提示し，挿入するコード補完機構を提案する．本機構で対象とする制御構文は，さしあたって `for` 文のみとするが，手法自体は `while` 文など他の制御構文に対しても適用することができる．プログラマの望んだコード片の見つけ方として，本機構では大きく分けて 2 段階の操作を経る．

(1) 既存のコード資産から作られたソースコードコーパス (以下，コーパス) から，型を使った検索により候補を探す．
(2) プログラマから与えられた条件を満たしているかをテスト実行をし，候補を絞り込む．

本研究では集合知を利用してコード片の検索を行う [7]．集合知とは，多くの人による知識や情報の統計のことである．検索エンジンのサジェストなどは集合知を利用しており，“多くの人が利用し

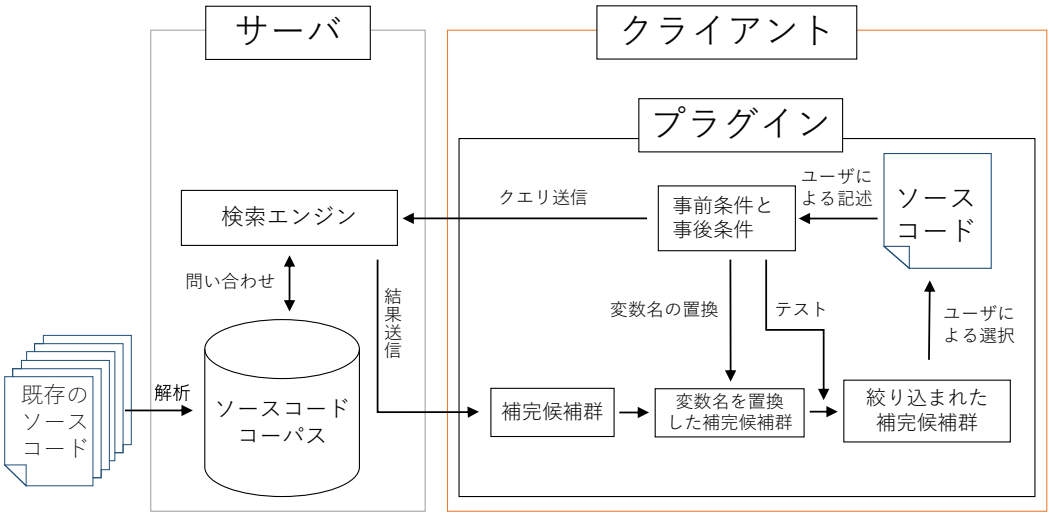

図 4　本機構の設計

ている情報は有益である” ということを活かして検索ワードを提案している．本機構でも同様に利用し，多くのプログラマが記述したコードは有益であると考え，プログラマの望むコードの中から多くの人が利用するコードを優先して提案する．これによりプログラマが望んでいるコード片の大まかな候補を見つける．そこからさらにテストの実行を行い候補を絞り込むことで，プログラマの望んだ動作をするコード片のみを見つけることをできる．

以降では，2.2 節で挙げた例を用いて，各動作の流れと各操作について説明する．

3.1　本機構の動作

本機構での動作の流れを図 4 に示す．本機構はクライアント・サーバ方式となっている．サーバ側には補完候補の検索システムが置かれおり，クライアント側には補完候補をさらに絞り込んでプログラマに表示するプラグインが置かれる．

まず，プログラマは for と記述した後にコード補完のコマンドを押す．するとポップアップウィンドウが表示されるので，プログラマは for 文の事前条件と事後条件を記述する．事前条件とは，for 文の実行前に成り立つべき条件，事後条件とは，事前条件を満たした状態で for 文を実行した後に満たされるべき条件のことである．2.2 節の例では，事前条件は「sum1 の値が 0」，「array1 の配列の中身は {1，2，3，4}」，事後条件は「sum1 の値が 10(first の総和)」というように具体例によって記述する．

これらの条件が入力されると，プラグインは事前・事後条件の情報をサーバの検索システムに送る．検索システムでは，プログラマが与えた条件に出現する変数の型情報のみを利用し，既存のコード資産からコードを検索する．その結果としてブロック内のコードを含む for 文全体が複数個得られ，サーバはこれをプラグインに返す．ここで返される for 文においては，中で使われる変数は抽象化されている．抽象化された変数は，プラグインにおいて，事前条件，事後条件内で使用した変数を用いて具象化される．更に，候補を絞り込むために，それぞれの for 文に対し，事前条件と事後条件を満たすかどうかを，実際に for 文を実行して調べる．最後に，条件を満たしているものだけをプログラマに提示する．

提示された候補がプログラマに選択されると，そのコードが図 5 の赤字のように挿入される．図 3(b) と比較すると，終了条件式で使用する配列が first になっており，ブロック内には first の総和を sum1 に代入する処理が記述されている．

```
1 boolean cmpSum(int[] first, int[] second){
2   int sum1 = 0, sum2 = 0;
3   for (int i = 0; i < first.length; i++){
4     sum1 += first[i];
5   }
6 }
```

図 5　本機構における for 文の補完例

このようにして，プログラマの望んだ機能を実装した制御構文が完成する．

3.2　コーパスの作成

本機構では，補完候補を検索するためのコーパスがサーバ側で必要である．コーパスの作成作業は，システムを構築する時にサーバで一度だけ行う処理である．コーパス作成処理では，過去に作成されたソフトウェアから， for 文の用例を抽出する．

コーパスには再利用元の既存のソースコードから for 文内で出現した変数の情報と，変数を抽象化した for 文のデータをまとめる．また，出現した変数は再利用の際に，プログラマが作成しているプログラムのスコープ内の変数に置き換えるため，変数名を抽象化し，抽象化した変数の型と番号を記録する．

コーパスは表 1 のようなテーブルとなっている．*id* 属性は，各タプルの id を表す属性である．値は 1 から順番にタプルごとに割り当てられる．使用回数属性は，解析中に現れた同内容のタプルの数を表す属性である．値が大きいタプルほど，より多くのソースコードで使用されているということである．グローバル変数属性は，for 文中で出現したグローバル変数のそれぞれの型の出現数を表している．ローカル変数はカウントしない．テンプレート属性は， for 文のデータを表す属性である．ブロック内の式などのデータはここに含まれる．また，ローカル変数を含めた各変数の抽象化の情報もここに含まれている．

例として図 6 の 2 つの for 文を抽象化する．図 6(a) では for 文中に出現する変数は i と n であり，

```
1 for(int i = 0; i < 10; i++){
2   n += i;
3 }
```

(a)

```
1 for(int i = 0; i < array.length; i++){
2   n += array[i];
3 }
```

(b)

図 6　抽出する for 文の例

どちらも int 型である．グローバル変数は n のみなので，“この for 文中で int 型のグローバル変数が 1 回出現した” ということを記録する．また，i を$1 に，n を$2 に抽象化する．このデータをコーパスに格納すると，表 1 の id の値が 1 のタプルになる．

同様に，図 6(b) では for 文中に出現する変数は i と array と n であり，i と n は int 型，array は int 型の配列である．グローバル変数は array と n なので，“この for 文中で int 型の配列のグローバル変数が 1 回，int 型のグローバル変数が 1 回出現した” ということを記録する．また，i を$1 に，array を$2 に，n を$3 に抽象化する．このデータをコーパスに格納すると，表 1 の id の値が 2 のタプルになる．

3.3　事前条件，事後条件の入力

本機構ではプログラマが “for” と入力した際，図 7 のようなポップアップが表示される．*Pre* 欄，*Post* 欄はそれぞれ事前条件を記述する欄，事後条件を表す欄である．Post 欄にはさらに値を記述する欄と，比較演算子を記述する欄がある．

プログラマはこのポップアップに for 文を実行したときの状態の変化を記述する．first の総和を求める場合は図 7 のように記述する．図 7 では，for 文の実行前では配列 first の中身が順に 1，2，3，4 で，sum1 の値が 0 となっている．ここで記述する条件とは，ソースコード中に静的に記述された条件ではなく，あくまでテストのための条件である．なので，ソースコード中の first や sum1

表 1　コーパスの中身

id	使用回数	グローバル変数				テンプレート [1]
		int_array	int	String	...	
1	10	0	1	0	...	`for(int $1 = 0; $1 < 10; $1++){$2 += $1;}`
2	20	1	1	0	...	`for(int $1 = 0; $1 < $2.length; $1++){$3 += $2[$1];}`
3	8	1	0	1	...	`for(int $1 = 0; $1 < $2.length; $1++){$3 += $2[$1];}`
4	12	1	1	0	...	`for(int $1 = 0; $1 < $2.length; $1++){$3 -= $1;}`
⋮	⋮	⋮	⋮	⋮	⋮	⋮

1　テンプレート属性の値は，実際には xml 形式で保存している

図 7　事前条件，事後条件の入力欄の例

の値が必ずしも Pre 欄で記述した状態と一致しなくてもよい．また，条件追加のボタンを押すと，右に Pre 欄，Post 欄がもう一つ現れ，複数の条件を記述することが可能となる．

この状態において for 文を実行した後の値を指定する条件として，図 7 では sum1 の値が，10 であることを示している．first には事後条件に何も記述されていないが，これは first に関する事後条件が無いことを示している．

プログラマによる入力が完了したら，サーバにデータを送り検索を行う．

3.4　検索

検索では，テーブル内の出現数属性をもとに検索を行う．プログラマから与えられた事前条件，事後条件内で記述された変数の型の出現数をもとに検索する．これは， for 文の前後で状態が変化する変数，すなわち事前条件，事後条件で記述された変数は， for 文内で使用されるはずであるからである．

図 7 の場合，int 型の配列が一つ (first) と int 型の変数が一つ (sum1) 記述されているという情報を利用する．入力のない型 (表 1 での String 型など) に関しては，出現数を 0 とする．したがって，「int 型の配列の出現数 1 回，int 型の変数の出現数 1 回，その他の型の出現数全て 0 回」として検索する．その結果，表 1 からでは id の値が 2 のタプルと 4 のタプルが，候補として返される．

3.5　抽象変数の具象化

検索の結果として得られた for 文には，抽象化された変数名が利用されているので，実際に利用するには，プログラムに合わせて抽象変数を実際の変数で具象化しなければならない．具象化する際に，その変数が for 文に対してグローバル変数かどうかによって，具象化の方法を変える必要がある．

`$1 = $2;` のように，変数宣言も無くいきなり利用されている`$1` や`$2` は，グローバル変数である．これらに対しては，プログラマが記述しているプログラム内で既に宣言されている変数で置換しなければならない．一方で，`int $3 = 0;` のように for 文内で変数宣言された`$3` は，グローバル変数ではなくローカル変数である．`$3` にはスコープ内で利用されていない新たな変数名を与えなければならない．

これに対し本機構では，グローバル変数は現在のスコープ内の変数で置換し，ローカル変数には別の名前を付けるようにする．また，グローバル変数を具象化する際に，プログラマがあらかじめ記述した事前条件，事後条件内で利用されている変数を利用する．これにより，プログラマが for 文内で利用する変数を指定することができる．例では，id の値が 2 の for 文と 4 の for 文に対し，

抽象化された int 型の配列と int 型の変数を，それぞれ first と sum1 で置換する．同じ型の変数が複数存在する場合は，それぞれの抽象変数の具象化のパターンが複数存在する．その場合は具象化の組み合わせの数だけ補完候補を生成する．

3.6 テストの実行

3.5 節で得られたそれぞれの補完候補に対して，事前条件を与えたうえで事後条件を満たしているかを調べる．このテスト実行はプラグイン側で行う．そのために，Java プログラムの単体テストを行うためのフレームワークである JUnit [8] を利用する．JUnit は一度に複数のテストを行う際，一部のテストでエラーが起こってしまっても実行を中断せずに全てのテストを行うことができる．

まず，プラグインは事前条件で与えられた入力を満たす式を挿入し，その後候補となる for 文を記述する．最後に，事後条件の assert メソッドを挿入する．assert メソッドは JUnit のライブラリから与えられており，引数にテストをしたい条件を boolean 型で与えることで，その内容を調べてテストが成功しているかを調べることができる．本機構では assert メソッドの引数として事後条件を与え，これにより事後条件を満たしているかを調べる．

最終的に，条件を満たしているものだけをプログラマに提示する．これにより，適切な変数を使い，プログラマの望んだ動作をする for 文を補完することができる．例では表 1 から検索した結果として id の値が 2 の for 文と 4 の for 文が返ってきたが，このうち id の値が 2 の方は配列の要素を変数に足しており，事後条件を満たしている．一方で id の値が 4 の方は，ループカウンタ変数で引いているので，事後条件を満たさない．従って，id の値が 2 の for 文が候補として提示される．

また，候補が複数存在した場合には，コーパスにおける使用回数属性の値が大きい方から提示する．

4. 実装

本機構は，サーバ側の検索器とクライアント側のプラグインに分けられる．サーバ側の実装はほぼ完了しているが，クライアント側は現在実装中である．本節ではそれぞれの実装について述べる．

4.1 コーパスの作成と検索

まず，コーパスを作成するために，既存のソースコードから for 文の情報を抜き出した．対象となるソースコードとして， GitHub の中の評価の高いプロジェクトの Java コードを使用した．for 文の抽出方法には，Eclipse で用意されている org.eclipse.jdt.core.dom ライブラリ を利用した．このライブラリは Eclipse で Java ソースコードを抽象構文木 (以下, AST) に変換するクラスのセットである．プログラムのそれぞれの構文要素は一つのオブジェクトとして扱われており，for 文は，ForStatement クラスのオブジェクトとして表現されている．本機構では既存のソースコードを AST に変換し，その中から ForStatement オブジェクトを抜き出た．

ForStatement クラスは for 文を構成する 4 つのフィールドを持っている．それぞれ初期化式，終了条件式，継続式，ブロック内のコードを表すオブジェクトである．フィールド内を見て出現した変数を探し，テーブルに追加して抽象化した．

テンプレート属性に挿入する情報は ForStatement クラスのインスタンスである．これを xml 形式にマーシャリングした文字列を挿入した．

コーパスの作成，管理に使用するデータベース管理システムとしては，MySQL を用いた． MySQL ではデータベースの管理を SQL クエリでの命令で行うので，コーパスの作成の際は CREATE 文でテーブルを作成し，それをコーパスとした．また，同データが出現したときに，該当タプルの使用回数属性の値を 1 増やすようにした．検索は，SELECT 文に WHERE 句でグローバル変数属性についての条件を与えることで絞り込んだ．グローバル変数属性の中で，事前・事後条件中に出現した型については，それらの型の出現数をそれぞれ条件として与えた．出現しない型については，それら型の出現数を 0 として条件を与えた．

検索でヒットしたタプルの中からテンプレート

属性の値を取り出した．

4.2 クライアント側のプラグイン

クライアント側は現在実装中なので，今後以下で述べるように実装を進めていく．

4.2.1 事前・事後条件の入力欄

プログラマが補完のコマンドを押した際，ソースコード中の不完全なコード片として “for” と記述されていた場合に，事前・事後条件を入力できるポップアップを出現させる．ポップアップが出現すると，フォーカスはエディタからポップアップへと移る．また，Tab キーを押すことで，カーソルを右隣のテキストエリアに移動させることができる．検索ボタン，条件追加ボタンにはそれぞれショートカットキーが割り振られており，マウスカーソルを使わずともそれらのボタンを押すことができる．

4.2.2 抽象変数の具象化

取り出したテンプレート属性の値は xml 形式にマーシャリングされているので，抽象変数名の具象化や，テスト実行を行うためにアンマーシャリングをする．これにより，コーパス作成時に挿入した ForStatement クラスのオブジェクトを復元する．その後，復元したオブジェクト内の抽象変数名を，型の対応する変数に具象化する．この時，同じ型の変数が複数出現した場合は，具象化の組み合わせの数だけ補完候補を生成する．

4.2.3 テストの実行

テストでは，事前に用意している JUnit テストプロジェクトにソースを書き込み，実行する．テストはメソッド単位で実行するので，候補となる for 文毎に別々のテストメソッドを作成する．各テスト用のメソッドは以下のように記述する．

(1) ポップアップの事前条件の記述欄から，状態を設定する記述をする．
(2) ForStatement オブジェクトから具象化した for 文のコードを取り出し，記述する．
(3) assert メソッドの引数として事後条件を記述する．

各候補に対応するメソッドの記述を全て終えたらテストを実行する．各テストにおいて，assert メソッドで true が返ったテストを，事前・事後条件を満たす候補とする．

4.3 補完候補の提示

コード補完プラグインでは，図 1 のような，実際にプログラマのエディタに表示される補完候補のリストを用意する必要がある． Eclipse では，各補完候補は ICompletionProposal インタフェースを実装したクラスのインスタンスで表される．このインタフェースを実装したクラスとして CompletionProposal クラスが存在する．テストが成功した際，CompletionProposal 型のインスタンスを生成し，for 文の内容をその中に入れる．これにより補完候補が完成する．

また，補完候補を列挙しているリストは List<ICompletionProposal>型となっている．候補が表示される際は，リストの先頭から表示されるので，使用回数属性が多い順に補完候補をリストに登録する．

プログラマが候補を選択すると，ブロックの中まで記述された for 文が挿入され，カーソルは for 文の次の行に移動するようにする．

5. 評価

本機構の性能として補完精度とコーディング速度の 2 点を評価する予定である．プログラマの望んだ for 文が補完出来るかを調べることで，補完精度が向上していることを確認し，既存の記述方法と比べて素早くコーディングが出来るかを調べることで，コーディング速度が向上していることを確認する．今後，以下のような方法で，評価を行う予定である．

5.1 精度

本機構で提示された候補が，本当に実際に利用されているかを調べる．方法としては，既存の Java コードの for 文を利用している場所で，本機構による for 文の補完を行う．実際に記述されたコードと同じコードが候補に現れるか，また，現れたときの表示順位を調べる．ただし実験の対象となる Java コードはコーパス作成時に利用しないもの

とする．

5.2 コーディング速度

次に，制御構文に対して，既存の記述方法を用いた場合と用いない場合とで，コーディング速度を比較する．既存の記述方法は 2 つあり，一つは制御構文に対する Eclipse の既存の補完機構を用いる方法，もう一つは制御構文の補完を用いず，手入力で記述する方法である．

ブロック内のコード量が多い場合は，本手法の方が記述時間が短くなることが予想されるが，コード量が極めて少ない場合は，逆に本手法を用いずに記述したほうが早い場合も考えられる．したがって，速度の比較方法として，いくつかの `for` 文を記述する際の記述時間を比べる．対象となる `for` 文のブロック内のコード量は様々なものを用意し，コード量毎のそれぞれの方法で記述した場合の時間を測定する．

6. 関連研究

ソフトウェア開発において，コーディングを支援するためにコード片を作成し，プログラマに提示する研究は多く行われている [9] [10] [11].

6.1 コードクローンの利用

コードクローンとは，コピー&ペーストなどにより再利用された既存のソースコードの断片のことである．この手法では，ソースコードを収集し，その中からプログラマからの要求を満足するであろうものを検索し，提示する．Steven らの研究 [9] では本研究と同様に，コードクローンを検索した後に，プログラマが与えた条件によって絞り込むことでコードを提示する．しかし，提示するコードはメソッド単位であり，グローバル変数やフィールドの値を変更するような処理をするメソッドを探すことができない．これに対して本研究では，提示するコードの単位は制御構文であり，変数の内容の変更が発生するコードも探すことができる．

6.2 プログラム合成

プログラム合成とは，作成したいプログラムの動作を絞り込むための条件を与え，その条件を満たすプログラムを合成する技術である．具体的には，プログラムに対する事前条件や事後条件などを与える．この技術を利用して，プログラマが望んだプログラムを提示する研究が行われている．Srivastava らの研究 [12] では事前条件，処理内容，事後条件の 3 つを記述することでプログラマの望むプログラムを合成する．これと比較すると，本研究では処理内容を記述せず，コーパスからの検索によってそれを得ている．それにより，プログラマの記述量を少なくすることができると期待される．

7. おわりに

本稿では，既存の IDE において制御構文を補完する際の問題点を提起し，その問題を解決するための機構の提案を行った．本手法では，既存のコード資産による検索と，テストの実行による候補の絞り込みを行う．これにより，ポップアップに事前条件と事後条件を与えるだけでプログラマの望んだコード片を提示することができる．

今後は，本機構の実装を進め評価を行う．また，その後の発展の方向として， `for` 文以外の制御構文への対応が考えられる．現在は `for` 文にしか対応していないので，他の構文にも対応させ，プログラマにより効率の良い入力をさせることが考えられる．

また，事前条件，事後条件の入力欄の簡略化が挙げられる．現在の実装ではプログラマに手入力をしてもらう点が，プログラマにとって手間である．したがって，よく使われる入力を選択できるようにするなどの改善が必要である．

謝辞 本論文を執筆するにあたり，方向性について重要な助言をいただきました岩崎英哉教授に深く感謝いたします． また，大日向大地様，浅野智之様，上田真史様には，論文に対して貴重なご意見をいただきました．心から感謝申し上げます．

参考文献

[1] Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using

the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[2] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic Parameter Recommendation for Practical API Usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.

[3] Code Recommenders. `http://www.eclipse.org/recommenders/`.

[4] Eclipse. `https://www.eclipse.org/`.

[5] NetBeans IDE. `https://ja.netbeans.org/`.

[6] IntelliJ IDEA. `http://www.jetbrains.com/idea/`.

[7] 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, 53(2):644–652, 2 2012.

[8] JUnit. `http://junit.org/junit4/`.

[9] Steven P. Reiss. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Zan Huang, Hsinchun Chen, and Daniel Zeng. Applying Associative Retrieval Techniques to Alleviate the Sparsity Problem in Collaborative Filtering. *ACM Trans. Inf. Syst.*, 22(1):116–142, January 2004.

[11] 島田 隆次, 市井 誠, 早瀬 康裕, 松下 誠, 井上 克郎. 開発中のソースコードに基づくソフトウェア部品の自動推薦システム A-SCORE. 情報処理学会論文誌, 50(12):3095–3107, 12 2009.

[12] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.