統合開発環境におけるコード補完の繰り返しに関する調査

A Study on Repetitiveness of Code Completion Operations on an Integrated Development Environment

大森 隆行* 桑原 寛明† 丸山 勝久‡

あらまし 近年の統合開発環境において，コード補完機能はなくてはならないものとなっている．コード補完機能により，開発者はコード入力作業を効率的に行うことができる．しかしながら，開発者が実際にどのようにコード補完を利用しているかについては，十分に明らかにされていない．本研究では，統合開発環境のコード補完機能の改善を目的として，統合開発環境上で開発者が行った操作の履歴からコード補完に関する操作を抽出し，コード補完がどのように行われているのかを調査した．この調査においては，同じコード片を挿入するコード補完操作の繰り返しに着目した．小規模なソフトウェア開発において得られた操作履歴を用いた実験において，基準となる補完操作と同じ文字列を補完する操作の行われた時刻の差を見たとき，基準操作から時間的に近いほど，補完操作全体に占める同じ文字列の補完操作の割合が高いという結果を得た．さらに，本論文ではこの結果に基づいたコード入力支援機能について考察する．

1 はじめに

統合開発環境 (IDE: integrated development environment) において，コード記述の効率を高めるため，コード補完機能はなくてはならないものとなっている．Java 言語の IDE として広く利用されている Eclipse では，コンテントアシストプラグインの一部としてコード補完ツールが実装されており，実開発においてよく利用されている．文献 [1] では，Java ソフトウェア開発者がどのように Eclipse を使用しているかを調査している．その中では，調査対象の全員がコンテントアシスト機能を利用しており，利用の頻度も，コードの削除，セーブ，次の単語への移動，ペーストに次いで，全コマンド中で 5 番目に高いと報告されている．このような統計的な分析は行われているが，開発者によって実際にどのようにコード補完が行われているか（補完対象や補完が行われるタイミングなど）については，これまでの研究ではほとんど明らかになっていない．

一方で，既存のコード補完の改善を試みる研究も近年行われている [2] [3] [4]．これらの手法では，既存のソフトウェアリポジトリに含まれるソースコードを元に API 使用パターンを明らかにしている．しかしながら，開発者が行った操作の履歴を利用したコード補完の改善手法は我々の知る限り，Robbes らの手法 [5] のみである．Robbes らは，IDE で記録された開発者のコード変更履歴がコード補完機能の改善に有効であると述べており，同時に変更が加えられた箇所や時間的な開発の切れ目等の情報を利用することで，補完の精度を改善している．本研究では，開発者がよく行う操作のパターンを考慮に入れたコード補完手法の実現を目指して，現状のコード補完機能がどのように使用されているのかを，IDE で行われた操作の履歴を利用して調査した．本研究では，メソッド以外を対象としたコード補完も対象としている点，コード補完操作の挿入文字列と時間差に基づいて頻出する操作のパターンを考慮し，そのパターンを対象とした入力支援手法を示している点が Robbes らの手

*Takayuki Omori, 立命館大学情報理工学部

†Hiroaki Kuwabara, 立命館大学情報理工学部

‡Katsuhisa Maruyama, 立命館大学情報理工学部

法とは異なる．

本研究では，これまでの我々の操作履歴分析の経験から，次の仮説を立てた．

仮説 あるコード補完操作（基準操作）と同じ文字列を挿入するコード補完操作は，基準操作に時刻が近いほど行われやすい．

この仮説の妥当性を調べるため，本研究では，あるコード補完操作（基準操作）と，それ以降に行われた各コード補完操作の時刻の差を調べた．そして，基準操作から 30 秒未満の時間帯，30 秒以上 60 秒未満の時間帯，60 秒以上 90 秒未満の時間帯，…というように，30 秒刻みで時間帯を設定し，各時間帯におけるコード補完操作の出現総数と，挿入文字列が基準操作と同じであるコード補完操作の出現数を計算した．

本研究におけるコード補完とは，Eclipse のコンテントアシスト機能によって識別子や予約語が自動的に入力されることを意味する．コード補完を実行する IDE の機能をコード補完機能，記録された IDE の操作のうちコード補完の実行に対応するものをコード補完操作と呼ぶ．Eclipse のリファクタリング機能や，コード生成機能（コンストラクタの生成，getter と setter の生成等）はコード補完操作に含まない．クラス名等を補完した際に自動的に `import` 文が挿入されることがあるが，この場合は `import` 文の挿入もコード補完操作の一部であるとみなす．

本研究では，OperationRecorder [6] により記録された操作履歴を用いて実験を行った．記録される操作履歴には，ソースコード上でのすべての編集履歴だけでなく，コードの編集がどのような操作によって行われたのか，いつ行われたのかの情報も含まれる．調査対象のコード補完操作を互いに比較し，補完による挿入文字列が同じものがあったとき，それらのコード補完操作を RCC(repetitive code completion operation) と呼ぶ．

我々は，RCC は開発者が定型的な作業を行っている場合に出現する可能性が高く，そのような操作は，IDE の支援により削減することが望ましいと考える．そこで，発見した RCC およびその前後の操作履歴を詳細に分析した．その結果，5 種類の頻出する RCC を特定できた．この分類は，RCC がどのような状況においてどの程度発生するのかを知るために役立つ．さらに，本論文では，各種類の RCC に起因するコード入力の負荷を削減するためのコード入力支援機能について考察を行う．

本論文の構成を述べる．2 章では，調査の詳細とその結果について述べる．この結果から，1 章で挙げた仮説が妥当であることを示す．3 章では，2 章の実験で発見された RCC を分類し，各分類項目に関して，RCC を削減するための新しいコード入力支援機能について考察を行う．4 章でまとめと今後の課題を述べる．

2 実験

本章では，まず操作履歴からコード補完操作を抽出する方法について説明する．次に，コード補完の総数と再出現回数の計算方法について述べる．その後，それらを実際の操作履歴に適用した結果について述べる．

2.1 手順

まず初めに，OperationRecorder により記録された操作履歴から，コード補完に関わる操作のみを抽出する方法について説明する．OperationRecorder は，Eclipse の Java エディタ上で行われた開発者の操作を記録し，操作履歴データを XML ファイルに出力するツールである．開発者は，操作の記録にあたって特別な操作を行う必要がないため，操作の記録が通常のプログラミングにおける作業に干渉することがない．ここでは，記録される操作（XML ファイルの要素）のうち，本研究に関係のあるものについて説明する[1]．なお，操作を行った時刻や開発者などの情報は，すべ

[1]現在のバージョンの OperationRecorder が出力する操作履歴ファイルの内容は下記ページにて説明されている．http://www.fse.cs.ritsumei.ac.jp/~takayuki/operec.html

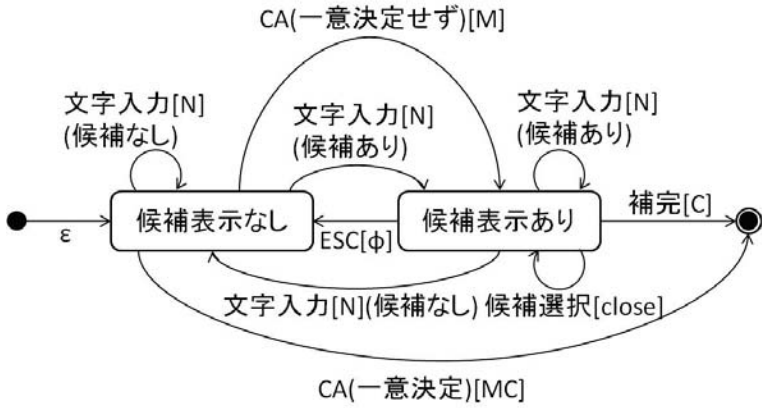

図 1　コード補完操作時の状態遷移

ての操作について利用できる．

- `normalOperation` 要素
  通常のコードの編集を表現する．操作対象のファイル，操作箇所を示すオフセット (ファイルの先頭からの文字数)，挿入された文字列，削除された文字列等を保持する．
- `compoundOperation` 要素
  Eclipse による自動編集を表現する．自動編集は，いくつかの文字列の挿入操作および削除操作により行われるので，自動編集による変更の内容を示すため，`compoundOperation` 要素は子要素として複数の `normalOperation` 要素（子操作と呼ぶ）を持つ．また，属性として，自動編集の種類を表現するラベルが保持される．リファクタリング等を行ったときには，ラベルの文字列によりリファクタリングの種類が示される．
- `menuOperation` 要素
  メニューバーのアイテム選択などによる Eclipse の機能の呼び出しを表現する．実行された Eclipse コマンドの ID がラベルとして保持される．

本論文では，操作履歴片を，上記操作に対応するアルファベット（それぞれ `N`, `C`, `M`）を `[]` で括って表現する．例えば，`[NMC]` という記述は，`normalOperation`, `menuOperation`, `compoundOperation` がこの順番でそれぞれ 1 つずつ出現する操作履歴片を意味する．

コード補完操作の抽出に先だって，関係のない要素はあらかじめ操作履歴から除去する．Eclipse のコード補完機能を使用したときに記録される `menuOperation` のラベルは `org.eclipse.jdt.ui.edit.text.contentAssist.proposals` [2]であり，`compoundOperation` のラベルは`"Typing"` [3]となるため，ラベルが一致しない操作をそれぞれ除去する．また，上記 3 種類以外の要素も除去する．

ここで，コード補完以外の `compoundOperation` を除去するため，コード補完が行われたときのエディタの状態遷移と記録される操作列の関係について考える．Eclipse におけるコード補完操作時の状態遷移を図 1 に示す．初期状態は，エディタ上で識別子や予約語の入力を開始していない状態である．状態には，補完候補が表示されていない状態，補完候補が表示されている状態があり，補完が行われると，終了状

[2]Eclipse 3.7.0 に標準で備わっている Java perspective のメニュー項目および Java エディタのコンテキストメニュー項目（OperationRecorder が記録対象としているもの）の中で，このラベルを持つものはコード補完機能のみである．

[3]英語環境の場合．また，コード補完操作以外においても，数多くの種類の `compoundOperation` がこのラベルを持つ．

態に遷移する．状態遷移を発生させる各イベントの後ろに付記されている [ ] 内のアルファベットは，そのイベントが発生したときに操作列に加えられる操作を表している．アルファベットと操作の種類の対応は前述の通りである．例えば，初期状態から文字を入力し補完候補が表示された後，補完の実行によって終了状態に遷移した場合，記録される操作は，[NC] となる．

ここで，normalOperation は，操作列に加わった時点ではその内容がまだ変更される可能性が残されていることに注意しなければならない．例えば，初期状態から文字'a' を入力することで，操作列には文字列"a"を挿入したことを表現する normalOperation が保持された状態になるが，次に文字'b' を入力すると，新たに normalOperaiton が追加されるのではなく，先の normalOperation の挿入文字列が "ab"に変化する．本論文では，変更される可能性が残されている normalOperation は，オープンな状態にあるという．変更される可能性のない normalOperation を，クローズな状態にあるという．図 1 における [close] という表記は，現在の操作列の末尾にある normalOperation をクローズな状態にすることを意味する．なお，図 1 における「文字入力」は，文字の入力操作のみでなく，削除操作も含んでいる．

図 1 におけるイベント ESC は，エスケープキーの押下等により，補完候補表示を消去することを示している．また，[Φ] は，操作履歴に変化を与えないイベントであることを示している．

CA は，Ctrl + Space キー等によるコード補完機能の呼び出しを示している．コード補完機能を呼び出したとき，補完の候補が 1 つしかない場合，Eclipse は補完候補の一覧を表示することなく，補完を実行する．

図 1 からわかる通り，1 回の補完を行うまでには，[N] や [M] が無数に出現する可能性がある．また，ESC により補完をキャンセルされた場合，操作履歴からキャンセルの実行を正確に検出することはできないため，コード補完に対応する操作列を正確に抽出することができなくなる．そこで，本研究では，頻出するパターンである，[MC]，[NMC]，[MNC] の 3 通りのみをコード補完操作として抽出することとする．[MC] は，1 文字も入力せずにコード補完機能を呼び出し，表示された候補から選択を行った場合，[NMC] は，文字を入力した後にコード補完機能を呼び出し，表示された候補から選択を行った場合，[MNC] は，コード補完機能を呼び出した後に文字を入力することで候補を絞り，その後選択を行った場合を意味している．Eclipse においては，コード補完時の compoundOperation は，それまでにキータイプにより入力した補完文字列の一部 (normalOperation として記録されている) を一度すべて削除し，改めて補完文字列を挿入するという操作となる．このため，上記のパターンの検出の際には，操作の種類のみを見るのではなく，compoundOperation の削除文字列と，直前の normalOperation の挿入文字列が一致することを確かめている．

ここでは，抽出した操作列をコード補完操作列と呼ぶ．最後に，抽出した各コード補完操作列内の compoundOperation に含まれる雑音を除去する．具体的には，compoundOperation の子要素の normalOperation に不要な文字の削除や挿入が含まれることがあるため，それらの操作を 1 つの挿入操作に統合する．また，各 normalOperation の挿入文字列の先頭と末尾の空白文字を取り除く．

ここで，図 2 の擬似コードに示した手順で，compoundOperation の出現数とそのうち再出現（以前のものと挿入文字列が同じ）であるものの出現数を計算する．おおまかには，compoundOperation の子操作すべての組み合わせを調べ，それらの操作の時間差ごとに，補完総数（当該時間帯においてコード補完が行われた回数）と再出現数（当該時間帯において行われたコード補完のうち，基準操作と挿入文字列が同じであるものの出現回数）を計算している．

図 2 中の各用語の意味は次の通りである．

- 配列オブジェクトの属性
  - size: 配列の長さを示す．

```
input: コード補完操作列の配列

 1: N : 各コード補完操作列に含まれる [C] の子操作である [N]
        すべてを連結した配列
 2: for(i=0; i<N.size; i++){
 3:   for(j=i+1; j<N.size; j++){
 4:     o1=N[i];  o2=N[j];
 5:     if(o1.cc==o2.cc) continue;
 6:     if(o1.insLen==0 && o2.insLen==0) continue;
 7:     td = o2.time-o1.time;
 8:     for(sec=0; sec<1800; sec+=30){
 9:       if(td < sec || sec <= td+30) continue;
10:       o1.cc.occTimes[sec/30]++;
11:       if(o1.insText == o2.insText){
12:         o1.cc.repTimes[sec/30]++;
13: }}}}
```

図 2　比較回数，再出現回数の算出手順

- normalOperation オブジェクトの属性
  - cc: 所属するコード補完操作列を示す.
  - time: 操作が行われた時刻（秒）を示す.
  - insLen: 挿入文字列の長さを示す.
  - insText: 挿入文字列を示す.
- コード補完操作列オブジェクトの属性
  - occTimes: 他のコード補完操作の出現回数を示す．長さ 60 の配列になっており，各時間帯ごとの出現回数を配列の各要素が示す.
  - repTimes: 再出現補完操作の回数を示す．長さ 60 の配列になっており，各時間帯ごとの出現回数を配列の各要素が示す.

図 2 の 5 行目は，1 つのコード補完操作列に含まれる複数の挿入文字列を互いに比較しないようにするための処理である．また，6 行目は，2 つの挿入文字列が互いに空文字列の場合に比較を行わないようにするための処理である．図 2 の 8 行目で，sec<1800 となっている理由は，2.2 で述べる実験の結果において，時間差 1800 秒を超える補完の出現回数が少なく，わずかな再出現回数の差異が再出現率に大きな影響を与えてしまうからである.

2.2　実験結果

2.1 で述べた手順を，著者のうち 2 名が実際にソフトウェア開発を行った際に記録された操作履歴に適用した．対象のソフトウェア開発は，小規模な Java プロジェクト 5 個分である．開発終了時点のソースコードは，合計で，73 ファイル，11425 行 (空行，コメント行含む) である．また，操作履歴ファイルのサイズは約 12MB，54551 個の操作 (XML 要素数) である.

各時間帯におけるコード補完操作列の総数（補完総数），再出現回数を合計し，時間帯ごとの再出現率を計算した．結果は図 3 に示す通りである．x 軸が時間 (秒)，y 軸 (左) が補完総数，再出現回数，y 軸 (右) が再出現率に対応している．図に示す通り，時間差が大きくなるにつれて，補完総数，再出現回数，再出現率のいずれの値も低下していく．コード補完操作が行われてから 30 秒未満の間にコード補完が行われた回数の合計は，1594 回，そのうち再出現であるものは 188 回（再出現率は 188/1594=11.8%）であった．再出現率は，30 秒以上 60 秒未満の時間帯では 7.8%，60 秒以上 90 秒未満の時間帯では 6.4%，以下，6.4%, 5.4%, 4.4% と続く．なお，時

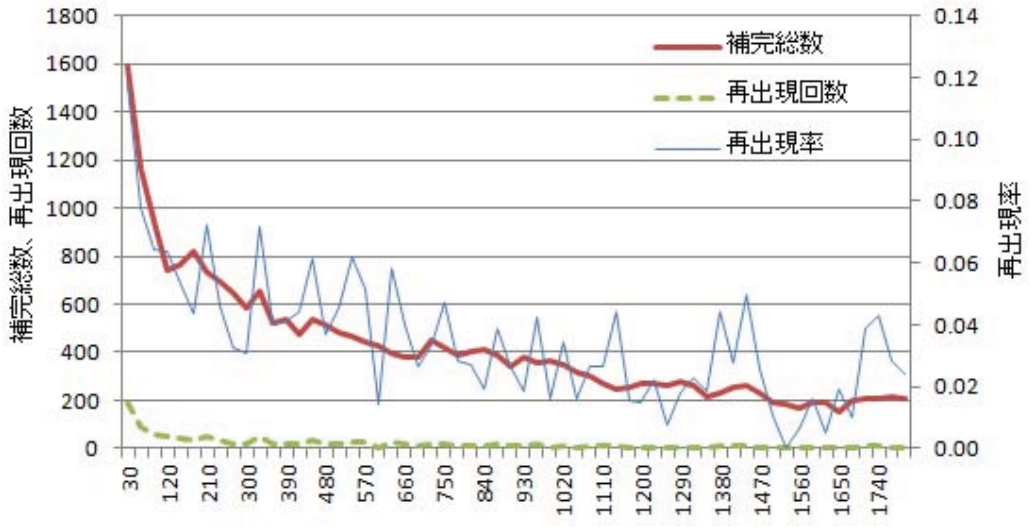

図 3　補完総数，再出現回数，再出現率

間と補完総数，再出現回数，再出現率との相関係数は，それぞれ–0.83，–0.62，–0.65であった．

上記の結果から，1 章で立てた仮説の通り，一度行われたコード補完と同じ文字列を挿入するコード補完操作は，時間が短いほど行われやすい傾向があると言える．

しかしながら，既存のコード補完機能を改善するために，補完候補推薦の戦略として,「直近の自動補完と同じ候補を示す」という手法は，上記の再出現率（30 秒未満に行われた補完のうち 11.8%しか再出現しない）を考慮すると，あまり効果的ではないと考えられる．そこで，3 章では，RCC をより詳しく調査した結果を示し，それに基づいてより有効なコード入力支援に関する考察を行う．

3　RCC の分類とコード入力支援の改善

3.1　RCC の分類

2.2 で用いた操作履歴において，RCC 前後の操作を OperationReplayer [7] を用いて調べた．OperationReplayer により，操作が行われた時刻のソースコードを復元し，操作を再生することができる．我々は，30 秒未満の時間帯において発生した 188 回の RCC について，手作業で当該補完操作およびその前後の操作を再生することで，RCC がどのような状況で発生するのかを調べた．その結果，表 1 に示す通り，RCC の出現状況として 5 通りの分類を得た.「補完対象」は，その分類のコード補完操作によって補完された対象を示している.「その他」には，開発者の入力ミスや，操作履歴の記録失敗によりソースコードを復元できなかった例等が含まれる．

以下，各分類について考察する．なお，説明で用いる例は，実例をベースにして作成したモックアップである．

(1) 類似した構造を持つ構文要素における編集

この種の RCC により入力されたコードの例を図 4 に示す．下線部はコード補完による入力を示す（図 5 以降も同様）．

(a) 行と (c) 行では，2 項演算子（<，>）のオペランドとして異なるオブジェクトの同じメソッドを呼び出している，また，(a) 行と (c) 行，(b) 行と (d) 行の組み合わせのように，`if` 文の条件節や分岐先の各ブロックには，類似した構造のコードが出現

表 1　RCC の分類

分類	補完対象	再出現回数
類似した構造を持つ構文要素における編集	主にメソッド	101
フィールドや変数の追加	主にフィールド	22
API 使用パターン	主にメソッド	11
定数への置換，使用	定数	8
ジェネリクスの型変数，`instanceof` 演算子の使用	型名	8
その他	—	38

```
if(o1.getTime() < o2.getTime()){          (a)
  ...
  Msg.print(debugString);                 (b)
}else if(o1.getTime() > o2.getTime()){    (c)
  ...
  Msg.print(debugString);                 (d)
}
```

図 4　類似した構造を持つ構文要素における編集

することが多い．我々の調査においては，このような類似構造を形成するコードを入力する際のコード補完操作では，多くの場合，補完の対象はメソッド名であった．

このようなコードを効率よく入力するためには，類似した構造の構文要素の組を発見し，その組を構成する構文要素において編集が行われた後，その対をなす構文要素での編集の際に，すでに行われたコード補完と同じコード補完を推薦することが有効であると考えられる．Java の制御構文に着目したとき，類似構造が出現しやすいものとして，(i) `if-else if-else` 文の条件節および分岐先の各ブロック，(ii) `switch-case` 文の各分岐先が挙げられる．また，複数のオペランドに同じ種類の値が使われることが多い演算子として，(iii) 剰余演算子，シフト演算子，複合代入演算子を除く 2 項演算子，(iv) 3 項演算子`?:`(第 2，第 3 オペランド) が挙げられる．

(2) フィールドや変数の追加

クラスに対して新しくフィールドを追加した場合や，メソッドに対して新しくローカル変数を追加した場合が，この分類項目に該当する．

図 5 に，この種の RCC により入力されたコードの例を示す．図 5 の例では，(a) 行のフィールド `newField` を手作業で（コード補完を使わずに）追加した後，(b) 行のコンストラクタの引数 `newField` の追加，(c) 行の代入文の追加，(d) 行の getter の追加，(e) 行の setter の追加が連続的に行われた結果，作成されたコードを示している．このとき，操作履歴上では，新しく追加されたフィールド `newField` およびコンストラクタの引数 `newField` が短い時間のうちに繰り返し補完されることになる．ここで，本研究ではコード補完により入力された文字列の同一性から RCC を抽出しているため，フィールド `newField` と，引数 `newField` の区別をしていないが，Java プログラムの慣習として，このようなケースでは両者同じ名前が使われることが多いため，このような RCC を発見することは補完戦略を考える上で有効であると考えられる．この分類項目より，次のようなコード補完戦略が有効であると考えられる．

> フィールド，変数，仮引数の宣言が加えられたとき，そのスコープ内において補完を行う際には，直近に宣言されたものほど候補の上位に表示する．

```
public class ClassA {
  ...
  private String newField;                          (a)
  public ClassA(String newField){                   (b)
    this.newField = newField;                       (c)
    ...
  }
  public String getNewField(){                      (d)
    return newField;
  }
  public void setNewField(String newField){         (e)
    this.newField = newField;
  }
}
```

図 5　フィールドや変数の追加

```
shell.setLayout(new BorderLayout());
shell.add(button, BorderLayout.CENTER);
```

```
StringBuffer buf = new StringBuffer();
buf.append("text: ");
buf.append(text);
```

図 6　API 使用パターン

ただし，宣言が加えられた時点から長い時間が経過した場合は上記の推薦対象から外すなどの方策についても検討すべきであると考える．

また，フィールドを追加した後，コンストラクタの拡張（図 5 の (b) 行の引数追加と (c) 行の代入文にあたるコードの追加）を自動的に行うコード生成機能も有効であると考えられる．現在の Eclipse では，すでに存在するフィールドに基づくコンストラクタの生成機能と getter および setter の自動生成機能が備わっているが，コンストラクタの拡張機能は存在しない．

(3) API 使用パターン

我々の実験では，図 6 に示すように，同じクラス名を繰り返し補完する必要があったり，同じメソッドを何度も呼ぶことが多い API の使用事例が見つかった．しかしながら，API 使用パターンに関しては，既存ソフトウェアリポジトリにおいて頻出するパターンを明らかにすることで API の推薦を行う手法が数多く提案されている [2] [3] [4] [8] [9] ため，本研究の改善対象外とする．

(4) `enum` 列挙子の参照

同じ `enum` 列挙型で宣言されている列挙子は，短い時間のうちに集中的に使用されることが多い．図 7 にこの種の RCC により入力されたコードの例を示す．

図 7 の (a) 行と (b) 行において，列挙型 `Sort` において宣言されている列挙子の出現が見られる．また，我々の調査においては，`enum` 宣言されている列挙子は，`if` 文

```
Sort getSort(){
  ...
  if(cond == 1){
    return Sort.A;        (a)
  }else if(cond == 2){
    return Sort.B;        (b)
  }
  ...
}
```

図 7　enum 列挙子の参照

```
List<Operation> array=new ArrayList<Operation>();
for(Operation op : array){
  if(op instanceof NormalOperation){
    NormalOperation o = (NormalOperation)op;
    ...
  }
}
```

図 8　ジェネリクスの型変数，instanceof 演算子の使用

の分岐先ブロックだけでなく，条件節においても集中的に使用されることが確認できた．さらに，列挙子の補完においては，単に列挙型名を繰り返し入力するのではなく，列挙子の一部を入力することで，列挙型の修飾も含めて補完することが多いことが確認できた．

このことより，次の補完戦略が有効であると考えられる．

直近に使用された列挙型の列挙子を補完候補上位とする．列挙型名による修飾の有無は，既入力部に倣うものとする．

(5) ジェネリクスの型変数，instanceof 演算子の使用

Java のジェネリクスや instanceof 演算子を使ったコードの場合，同じ型名を短時間の間に繰り返し入力することが多い．この種の RCC により入力されたコードの例を図 8 に示す．図 8 では，Operation 型の要素を持つ ArrayList を生成し，for-each 文を使用して各要素について処理を行っている．ループの中では，着目している要素が NormalOperation 型（Operation を継承）であるかをチェックし，NormalOperation 型の変数を宣言してから処理を行っている．この例のように，ジェネリクスを使ったコードの場合，型引数（例では Operation）が集中して出現することが多いと考えられる．また，instanceof 演算子による型チェックとキャストの組み合わせも出現することが多い．

この事例より，コレクションインスタンスと for-each 文が組み合わせて使われる場合に，コレクションを構成する要素の型の名前と変数，コレクションインスタンス名を自動的に補完することが望ましいと考えられる．ここで，実際の入力時（"for("まで入力した時点）には for 文と for-each 文の区別はつかないが，現在のスコープにおいて参照可能なコレクションインスタンスがある場合，その要素の型を補完候補とすることは妥当であると考えられる．現在の Eclipse では，for 文の括弧内（括弧の入力直後）でコード補完機能を呼び出しても，そのような候補が

優先的に表示されない．

また，`instanceof` 演算子を `if` 文の条件節で使用したとき，続くブロック内にその型の変数を宣言し，適切なキャスト，代入を行う文を生成するコード生成機能も有用であると考えられる．

4 おわりに

本論文では，IDE における，繰り返しのコード補完操作 (RCC) が比較的短い時間のうちに発生することが多いことを，開発者の操作履歴を用いた実験により示した．さらに，RCC の発生状況を分類し，各分類に関して，新しいコード入力支援手法を示した．本論文で示したコード入力支援手法により，普通にコード入力を行えば RCC が発生してしまう状況において，既存手法より効率良いコード入力が期待される．

今後の課題について述べる．まず，本研究で提案したコード入力支援手法を実装し，RCC に関するコード入力支援の性能を評価する必要がある．また，本研究によって，2.2 で述べた，時間差と補完総数，再出現回数，再出現率の相関に関してすべての要因を明らかにできたわけではない．3 章では，30 秒未満の時間帯にのみ着目したが，時間帯が異なれば，出現する RCC の種類に違いが発生する可能性もある．コード補完操作の時間差と RCC の出現との関連については，引き続き調査を進める必要がある．さらに，本研究では，コード補完操作のみを対象とした実験を行ったが，今後，一般的なトークン入力を扱う実験を行う予定である．この実験により，コード補完を行うかどうかに関わらず，同一識別子への参照が短時間のうちに繰り返し挿入される傾向があるかどうかを明らかにすることを目指す．単なる操作の時間差のみではなく，コード補完操作により追加された変数のスコープや，開発のタスク，セッション [10] の切れ目を考慮した調査，多様な開発者による大規模なソフトウェア開発を対象とした実験により，得られた結果の普遍性を確かめることも今後の課題である．

謝辞 本研究の一部は，科研費（24500050, 24700034, 24700036）の助成による．

参考文献

[1] G. C. Murphy, M. Kersten, and L. Findlater, “How Are Java Software Developers Using the Eclipse IDE?,” IEEE Software, vol. 23, issue 4, pp. 76–83, 2006.
[2] M. Bruch, M. Monperrus, and M. Mezini, “Learning from Examples to Improve Code Completion Systems,” Proc. ESEC-FSE ’09, pp. 213–222, 2009.
[3] D. Hou and D. M. Pletcher, “An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion,” Proc. ICSM ’11, pp. 233–242, 2011.
[4] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, “Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion,” Proc. ICSE ’12, pp. 69–79, 2012.
[5] R. Robbes and M. Lanza, “Improving Code Completion with Program History,” Automated Software Engineering, vol. 17, no. 2, pp. 181–212, 2010.
[6] T. Omori and K. Maruyama, “A Change-aware Development Environment by Recording Editing Operations of Source Code,” Proc. MSR ’08, pp. 31–34, 2008.
[7] T. Omori and K. Maruyama, “An Editing-operation Replayer with Highlights Supporting Investigation of Program Modifications,” Proc. IWPSE-EVOL ’11, pp. 101–105, 2011.
[8] M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining API patterns as partial orders from source code: from usage scenarios to specifications,” Proc. ESEC-FSE ’07, pp. 25–34, 2007.
[9] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “MAPO: Mining and Recommending API Usage Patterns,” Proc. ECOOP ’09, pp. 318–343, 2009.
[10] R. Robbes and M. Lanza, “Characterizing and Understanding Development Sessions,” Proc. ICPC ’07, pp. 155–166, 2007.