THE IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS (JAPANESE EDITION)

電子情報通信学会 論文誌 D 情報・システム

VOL. J99-D NO. 4
APRIL 2016

情報・システムソサイエティ

一般社団法人 電子情報通信学会
THE INFORMATION AND SYSTEMS SOCIETY
THE INSTITUTE OF ELECTRONICS, INFORMATION AND COMMUNICATION ENGINEERS

論　文

書き忘れたコードに対するコード補完について

石原　知也[†a)]　　肥後　芳樹[†b)]　　楠本　真二[†c)]

On the Code Completion for Missing Middle Code

Tomoya ISHIHARA[†a)], Yoshiki HIGO[†b)], and Shinji KUSUMOTO[†c)]

あらまし　ソースコードの実装を支援する方法の一つとしてコード補完が挙げられる．近年，既存コードから収集した情報を利用してコード補完を行う手法が多数提案されている．これらの手法は書きかけのコードに続くコードを補完できるが，書きかけのコードの途中に書き忘れがあった場合にそれらを補完できない．本論文では，そのような書き忘れが発生する頻度について調査した．また，書き忘れたコードも補完対象とすることでコード補完の機会がどの程度向上するのかについても調査した．これらの調査においては，被験者実験を行った．実験では，被験者が行った計 51 タスクのうちの 41 タスクにおいて書き忘れが発生していた．また，その 41 タスクのうちの 10 タスクについては，既存コードから収集した情報を利用することにより書き忘れたコードに対して適切なコードを提示できることを示した．

キーワード　コード補完，コードクローン検出，ソースコード解析

1. まえがき

コードの補完技術を利用することで効率的に必要なコードを実装できる．コードの補完とは，開発者が必要としているコードを，（半）自動的にエディタ上に生成することである．コードの補完技術を利用することで，開発者は必要なコードを直接記述する必要がなくなる．現在では，コードの補完機能は Eclipse，IntelliJ IDEA，Microsoft Visual Studio 等の統合開発環境では標準的に装備されており，多くの開発者がコードの補完機能を利用しているとの調査結果もある [1]．統合開発環境がもつコードの補完機能は比較的単純であり，書きかけの変数名やメソッド名の補完や for 文等の制御構造のテンプレートを生成するのみである．そのため，より有益なコード補完を実現するための研究が行われている [2]〜[12]．

統合開発環境に装備されている補完機能は利用する変数名の補完や呼び出すメソッドの名前の補完（字句単位での補完）であるのに対して，これらの手法はプログラム文単位でのコード補完や必要なメソッド呼び出しの予測を行う．既存手法を用いることにより，より効率的に必要なコードを実装できる．しかし，既存のコード補完手法は，書きかけのコードに書き忘れがあった場合に，その書き忘れたコードを補完できない．開発者は必要なコードを書き忘れてしまう場合があるかもしれない．そのため，書き忘れたコードも補完対象にしたコードの補完機能を利用することで，より効率的にコードの実装を行えると著者らは考えた．

本論文では，コードの書き忘れが発生する頻度を調査した．また，書き忘れたコードに対してどの程度コード補完を行えるかを調査した(注1)．これら二つの調査は，被験者実験によって行われた．各被験者にはコードを実装するタスクが与えられ，被験者がコードを実装する様子は画面を動画として撮影することにより記録した．被験者のタスクが全て終了した後に，著者らがその動画を閲覧することによって調査を行った．その結果，全 51 タスクのうち 41 タスクにおいて書き

† 大阪大学大学院情報科学研究科，吹田市
Graduate School of Information Science and Technology, Osaka University, Yamadaoka 1–5, Suita-shi, 565–0871 Japan
a) E-mail: t-ishihr@ist.osaka-u.ac.jp
b) E-mail: higo@ist.osaka-u.ac.jp
c) E-mail: kusumoto@ist.osaka-u.ac.jp
DOI:10.14923/transinfj.2015JDP7086

(注1)：本論文における二つの調査は，文献 [13] にて既発表である．しかし，文献 [13] では，調査方法の概要と大まかな実験結果を報告しているのみである．それに対して本論文では，調査方法の詳細な説明，実験結果の詳細な説明及びその考察，関連研究等についても記述している．

忘れが発生していた．また，全 51 タスクのうち 32 タスクにおいて，既存コードから収集した情報を利用することで適切なコードを提示できることがわかった．また，それら 32 タスクのうちの 10 タスクでは書き忘れたコードに対する適切なコードが存在していた．

本論文の主な貢献は以下の三つである．

- 被験者実験を行い，コードの書き忘れが実際に発生することを確認した．
- 書き忘れたコードも補完対象にすることによって，コード補完の機会が増えることを確認した．
- 本論文で考案した調査手法は，書きかけのメソッドを与えると自動的に補完候補となりうるプログラム文を検出する．そのため，書き忘れたコードに対するコード補完手法の一部として利用可能である．しかし，コード補完手法を完成させるためには，複数の候補が存在した場合のフィルタリングや順位付け等についても考案する必要がある．

2. 準　　備

本章ではまずこの研究の動機について述べ，そのあとに本論文で用いる用語を定義する．

2.1 研究の動機

図 1 (a) は書きかけの Java メソッドである．このメソッドにおいて，開発者は引数で与えられた四角形の左上隅の X 座標と Y 座標を取得するコードを記述している．次に開発者はこの四角形の右下隅の X 座標と Y 座標を取得し，それらを String 型に変換し，メソッドの返り値として指定するコードを記述しようとしている．このような状態において，もし開発者が既存のコード補完手法を利用すると，書きかけのコードに続くコード（図 1 (b) の吹き出し内のコード）が開発者に提示される．開発者は，提示されたコードが自分の意図したコードであれば，それを選択するのみでエディタ上にそれを反映させることができる．この例では，右下隅の X 座標と Y 座標を取得するプログラム文と，座標を String 型に変換しメソッドの返り値として指定するプログラム文が補完され，これらを開発者が自分自身で記述する必要がなくなった．

```
private String getRectCoords(Rectangle2D rectangle){
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();
}
```

(a) 書きかけのコード

```
private String getRectCoords(Rectangle2D rectangle){
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();
    int x2=x1 + (int)rectangle.getWidth();
    int y2=y1 + (int)rectangle.getHeight();
    return x1 + "," + y1+ ","+ x2+ ","+ y2;
}
```

(b) 書きかけのコードに続くコードを補完対象にした場合

```
private String getRectCoords(Rectangle2D rectangle){
    if(rectangle == null) {
      throw new IllegalArgumentException(
            "Null 'rectangle' argument.");
    }
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();
    int x2=x1 + (int)rectangle.getWidth();
    int y2=y1 + (int)rectangle.getHeight();
    return x1 + "," + y1+ ","+ x2+ ","+ y2;
}
```

(c) 書き忘れたコードも補完対象にした場合

図 1　補完されるコードの例
Fig. 1　Example of completed code.

書きかけのコードに続くコードだけではなく，書き忘れたコードも補完対象にすることによって，コード補完はよりいっそう実装を支援できると著者らは考えた．例えば，Java の場合ではプログラムの実行時に NullPointerException が発生して，使用している変数に対して null かどうかのチェックを忘れていることに初めて気づく場合が多い [14]．図 1 (c) の上部の吹き出し内のコードは，引数が null かどうかのチェックをしている．このように書き忘れたコードも補完対象とすることによって，コード補完が実装に寄与する機会が向上すると期待できる．

以上のことから，本論文では，書き忘れたコードの補完に関連する以下の項目について調査を行う．

調査項目 1　どの程度の頻度でコードの書き忘れは発生するのか．

調査項目 2　書き忘れたコードも補完の対象にすることで，コードの補完が実装に寄与する機会はどの程度向上するのか．

2.2 用　　語

ここでは，本論文で用いる用語の定義を行う．**コード**とは，一つ以上の連続するプログラム文を表す．**書き忘れたコード**とは，実装済みのコードには存在していないがプログラムが正しく動作するためには必要なコードのうち，実装済みのコードよりも上部若しくは内部に挿入されるべきコードを指す．コード補完を行いたいメソッドを**補完対象メソッド**と呼ぶ．コード補完手法によって開発者に提示されたプログラム文を**補完候補**，またその補完候補を提示する元となったメソッドを**補完候補元メソッド**と呼ぶ．

これらの定義を図 1 に適用すると次のようになる．

```
 1 private String getRectCoords(Rectangle2D rectangle){
 2   if(rectangle == null) {
 3     throw new IllegalArgumentException(
 4               "Null 'rectangle' argument.");
 5   }
 6   int x1=(int)rectangle.getX();
 7   int y1=(int)rectangle.getY();
 8   int x2=x1 + (int)rectangle.getWidth();
 9   int y2=y1 + (int)rectangle.getHeight();
10   return x1 + "," + y1+ ","+ x2+ ","+ y2;
11 }
```

図 2　図 1 に示す補完候補の補完候補元メソッド
Fig. 2　The method for suggesting candidate statements to the method in Fig. 1.

メソッド getRectCoords は補完対象メソッドである．図 1 (b) 及び図 1 (c) では，複数のプログラム文が開発者に提示されている状態である．これらは補完候補である．図 2 に含まれるプログラム文が図 1 (b) 及び図 1 (c) において補完候補となっている．このときに図 2 に示すメソッドを補完候補元メソッドと呼ぶ．図 1 (c) において，上部の吹き出しに含まれるコードは書き忘れたコードであるが，下部の吹き出しに含まれるコードは書き忘れたコードではない．

図 1 (a) と図 2 に示すように，補完対象メソッドに含まれている全てのプログラム文は，補完候補元メソッドにも含まれている．言い換えれば，補完候補元メソッドと補完対象メソッドは Type-3 クローン(注2)であるといえる．本論文では，補完候補元メソッドは補完対象メソッドの全てのプログラム文を含むことから，補完候補元メソッドは補完対象メソッドの**スーパークローン**であるという．本論文の調査では，補完対象メソッドのスーパークローンを検出することによって，書きかけのコードに対する補完候補の有無を判断する．

3. 補完候補の有無の自動判定方法

本論文の調査では，補完対象メソッドに対して，適切な補完候補の有無を判断した．補完候補の有無はソースコードを解析することにより自動的に行った．補完候補が存在していた場合，それが適切かどうかの判断は著者らが目視により行った．

本章では，自動判定方法について説明する．この自動判定方法は，入力として補完対象メソッドを受け取り，そのメソッドのスーパークローンを出力する．スーパークローンが存在していない場合はなにも出力しない．出力されたスーパークローンには存在しているが入力として与えたメソッドには存在していないプログラム文が補完候補である．

(注2)：Type-3 クローンとは，プログラム文単位での追加，削除，または変更を含んだクローンである [15].

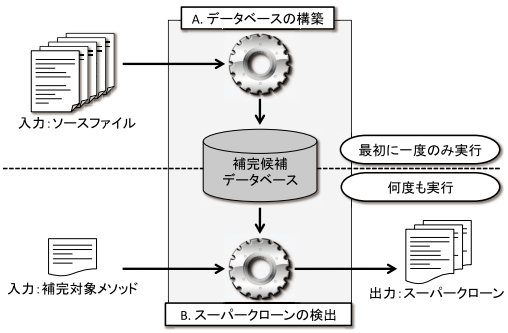


図 3　補完候補の有無を判定する方法の概要
Fig. 3　Overview of the way we decide whether or not there are candidate statements for a given Java method.

この自動判定方法は，書きかけのコードに続くコードと書き忘れたコードのどちらにも対応している．図 3 はその概要を表している．まずは，補完候補データベースを構築する必要がある．データベースを構築するために，大量のソースファイルを入力として与える．与えられたソースファイルは個別に解析され，その中に含まれているメソッドの情報が補完候補データベースに格納される．そして，補完対象メソッドが与えられると，データベースに格納されているメソッドとのスーパークローンを検出することにより，補完候補の有無を判定する．

この判定は二つの処理からなる．

処理 A　データベースの構築

処理 B　スーパークローンの検出

処理 A では，既存のソースファイル群に含まれる各メソッドをプログラム文の列に変換する．各プログラム文からはハッシュ値が計算される．ハッシュ値はデータベースに格納される．

処理 B では，補完対象メソッドがプログラム文の列に変換され，各プログラム文からはハッシュ値が計算される．次にその情報を用いてデータベースに問い合わせを行い，補完対象メソッドのスーパークローンを検出する．スーパークローンが検出された場合は，補完候補が存在していることを意味する．

以降，本章では，二つの処理について詳細に説明する．

3.1 処理 A データベースの構築

図 4 は処理 A の概要を表している．この図に示す

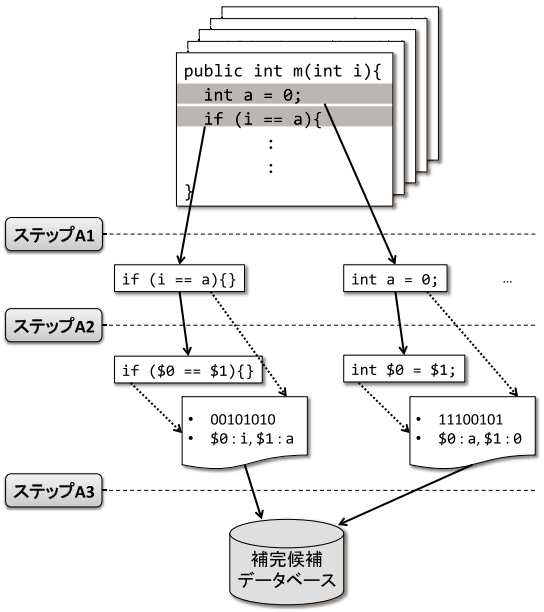


図 4　処理 A の概要
Fig. 4　Overview of method analysis.

ように，メソッドを解析して補完候補データベースを構築する手順は次の三つのステップからなる．

ステップ A1　与えられたソースファイル群に含まれる各メソッドをプログラム文の並びに変換する．

ステップ A2　各プログラム文の中に含まれる変数名及びリテラルを正規化する．

ステップ A3　各プログラム文の情報をデータベースに登録する．データベースに登録される情報は，プログラム文のハッシュ値と正規化前後の字句の対応関係である．

以降，各ステップについて詳しく説明する．

ステップ A1 メソッドをプログラム文の並びへ変換

与えられたソースファイル群から抽象構文木を構築することによってメソッドを抽出する．抽出したメソッドはプログラム文の並びへ変換される．抽出されたメソッドは抽象構文木の部分木であるため，メソッドの中に含まれるプログラム文の特定は抽象構文木をたどることによって簡単に行える．各プログラム文には固有の ID が割り振られる．この判定方法は Java 言語を対象としており，プログラム文を以下の 3 種類に分類する．

単文　セミコロン（“;”）で終了している文を指す．例えば，変数宣言文や return 文が挙げられる．また，switch 文の case エントリもこのカテゴリーに含まれる．

ブロック文　メソッド内において，ブロック（“{”と “}” で囲まれた範囲）をもつ文．例えば，if 文や for 文等が挙げられる．

シグネチャ文　メソッドのシグネチャも文として扱う．Java 言語のシグネチャは，返り値の型，メソッド名，仮引数で構成されている．

図 2 において，1 行目はシグネチャ文，2 行目はブロック文，3〜4 行目は単文となる．

ステップ A2 プログラム文の正規化

このステップでは，各プログラム文に対して正規化を行う．正規化の対象は変数とリテラルである．正規化処理として *Parameterized Matching* [16] を用いる．*Parameterized Matching* は同じ変数は同じ文字列を置換に使用し，異なる変数に対しては異なる文字列を用いて置換を行うという正規化方法である．例えば，図 2 の 8 行目の文に対して *Parameterized Matching* を適用すると，以下のように正規化される．

```
int x2=x1 + (int)rectangle.getWidth();
        ↓正規化
int $0=$1 + (int)$2.getWidth();
```

ステップ A3 プログラム文をデータベースへ登録

このステップでは，正規化された各プログラム文の以下の情報をデータベースに格納する．

ハッシュ値　ハッシュ値は，正規化後のプログラム文の文字列に対して，MD5 アルゴリズムを適用することによって計算される．

正規化前後の字句の対応関係　例えば，以下のプログラム文の正規化では，変数 a が$0 へ変換されたという情報と，リテラル 0 が$1 に変換されたという情報がデータベースに格納される．

```
int a = 0; → int $0 = $1;
```

3.2　処理 B スーパークローンの検出

処理 B では，与えられた補完対象メソッドのスーパークローンを検出する．スーパークローンが検出された場合，与えられた補完対象メソッドに対して補完候補が存在していることになる．スーパークローンを検出する手順は次の三つのステップからなる．

ステップ B1　補完対象メソッドに含まれるプログ

```
1  public void calculate(){
2    int z = 0;  -- A
3    int x = read(); -- B
4    int y = read(); -- C
5    z = y - x; -- D
6    println(z); -- E
7 }
```

(a) 補完対象メソッド

```
1  public void calculate(){
2    int c = 0; -- A
3    int a = read(); -- B
4    int b = read(); -- C
5    a = 4 * a + 5;
6    c = b - a; -- D
7    println(a,b);
8    println(c); -- E
9 }
```

(b) スーパークローン

```
1  public void calculate(){
2    int c = 0; -- A
3    int a = read(); -- B
4    int b = read(); -- C
5    a = 4 * a + 5;
6    println(a,b);
7    println(c); -- E
8    c = b - a; -- D
9 }
```

(c) 条件 A を満たすメソッド

```
1  public void calculate(){
2    int c = 0; -- A
3    int a = read(); -- B,C
4    int b = 4 * a + 5;
5    c = b - a; -- D
6    println(a,b);
7    println(c); -- E
8 }
```

(d) 条件 B を満たすメソッド

```
1  public void calculate(){
2    int c = 0; -- A
3    int a = read(); -- B
4    int b = read(); -- C
5    a = 4 * a + 5;
6    c = a - b; -- D
7    println(a,b);
8    println(c); -- E
9 }
```

(e) 条件 C を満たすメソッド

図 5　補完対象メソッド，スーパークローン，及びスーパークローンにならないメソッドの例（(b) から (e) は補完候補データベースに登録されているメソッドである）

Fig. 5　Example of target method for code completion, its super clone, and methods that are not regarded as super clones (methods from (b) to (e) are stored in complement candidate database).

ラム文からハッシュ値を計算する．そして，それら全てのハッシュ値を含むメソッドをデータベースから取得する．

ステップ B2　補完対象メソッド内のプログラム文とデータベースから取得したメソッドに含まれるプログラム文をハッシュ値を用いて対応付けを行う．データベースから取得したメソッドが下記のいずれかの条件を満たす場合，そのメソッドは破棄される．

条件 A　データベースから取得したメソッド内のプログラム文の並び順が，補完対象メソッドに含まれるプログラム文の並び順と異なる．

条件 B　データベースから取得したメソッド内のいずれかのプログラム文が，補完対象メソッド内の二つ以上のプログラム文と対応する．

ステップ B3　対応付けされたプログラム文において，変数とリテラルの対応付けを行う．データベースから取得したメソッドが下記の条件を満たす場合，そのメソッドは破棄される．

条件 C　変数やリテラルの対応付けに不整合が生じる．

ステップ B3 を通過したメソッドが補完対象メソッドのスーパークローンである．

図 5 はステップ B2 若しくはステップ B3 で破棄されるメソッドの例を示している．この図において，プログラム文の後のアルファベットは図 5 (a) と図 5 の他のメソッドとの対応関係を表すラベルである．もし，開発者が図 5 (a) に示すメソッドを補完対象メソッドに指定すると，図 5 (b) に示すメソッドがスーパークローンとして検出される．それ以外の三つのメソッド（図 5 (c), 5 (d) 及び 5 (e)）はスーパークローンにならない．

図 5 (c) は条件 A を満たすメソッドである．補完対象メソッド（図 5 (a)）の 5 行目及び 6 行目が，このメソッドの 7 行目及び 8 行目に対応するが，そのソースコード上での順番が入れ替わっている．よってこのメソッドはスーパークローンとして検出されない．

図 5 (d) は条件 B を満たすメソッドである．このメソッドの 3 行目が補完対象メソッドの 3 行目と 4 行目の二つのプログラム文と対応している．よって，このメソッドもスーパークローンとして検出されない．

図 5 (e) は条件 C を満たすメソッドである．このメソッドの 3 行目，4 行目，及び 6 行目が，対象メソッドの 3 行目，4 行目，及び 5 行目と対応する．しかし，これらのプログラム文には変数の対応関係において不整合が生じている．図 5 (a) では，5 行目のプログラム文の最後の変数は x であり，これは 3 行目のプログ

ラム文で代入が行われている（ラベル B）．その一方で，図 5 (e) では，6 行目の最後の変数は b であり，これは 4 行目のプログラム文で代入が行われている（ラベル C）．以上のことより，ラベル D のプログラム文は変数の対応関係に不整合が生じているために，スーパークローンとして検出されない．

4. 実　　験

2.1 で示した二つの調査項目について回答するため，被験者参加型の実験を行った．この実験では，UCI データセット(注3)を利用して補完候補データベースを構築した．UCI データセットは Java で記述されたオープンソースソフトウェアの集合であり，13,192 のプロジェクト，2,127,877 のソースファイル，20,449,986 のメソッドが含まれている．

調査項目に回答するためには，いつ書き忘れが発生したのかを特定する必要がある．本実験では，被験者が実装済みのコードの間に新しくコードを記述した場合，その記述したコードを書き忘れたコードとした．図 6 は書き忘れたコードを示す例である．コードの左の数字は，記述された順番を表している．この例では，3 番目に記述されたコード（引数の null チェック）は，1 番目と 2 番目に記述されたコードよりも，ソースコード上において上部にある．この場合は，3 番目に記述されたコードを書き忘れたコードとする．

なお，既に記述されたコードの一部が変更された場合には書き忘れコードとはしない．例えば図 6 において，全てのコードが実装された後に 2 番目に記述されたコードが修正された場合でも，そのコードは書き忘れコードにはならない．

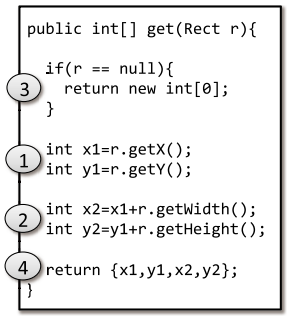


図 6　コードの記述順序の例
Fig. 6　An example of code writing order.

4.1　実験の方法

本実験は以下の手順で行った．

手順 1　被験者は与えられた仕様に基づいて Java のメソッドを実装した．被験者が実装している画面は動画として録画された．

手順 2　**UCI** データセットを利用して著者らが補完候補データベースを構築した．

手順 3　著者らは録画した動画を調査することによって，二つの調査項目に回答した．

手順 1 では，被験者は与えられた仕様に基づいて Java のメソッドを実装した．実装は画面を動画として録画する設定を行ったワークステーションにおいて行った．画面の録画には，*CamStudio*(注4) というソフトウェアを利用した．そのワークステーションには Windows Server 2008 R2 がインストールされており，被験者はリモートデスクトップ接続でそのワークステーションにログインした．

Java メソッドの実装において，著者らは被験者には特に制限を課さなかった．被験者は普段通りに Java メソッドの実装を行った．また，被験者が実装したコードが仕様を満たしているかどうかを確認するために，著者らはあらかじめ各タスクについてテストケースを準備した．被験者の実装した Java メソッドが用意された全てのテストケースを満たしたときに，著者らは被験者が与えられた仕様を満たす Java メソッドの実装を終えたとみなした．

手順 2 では，著者らが **3.1** で紹介した処理 A を実行してデータベースを構築した．データベースの構築には約二時間を要した．

手順 3 では，手順 1 で録画した動画を著者らが調査した．調査項目 1 へ回答するために，どのような順番でプログラム文が実装されていくのかを調査した．また，調査項目 2 へ回答するために，動画の中の被験者の実装の手順を手元で再現した．そして，補完候補データベースに問い合わせを行い，スーパークローンを検出した．検出されたスーパークローンを著者らが手作業で調査し，被験者の完成形のコードに含まれるプログラム文が存在しているかを判断した．存在していた場合には，適切な補完候補が存在していた，として扱った．この調査では，書きかけのコードを用いたデータベースへの問い合わせを，被験者が新たにプログラム文を実装するたびに行った．

(注3)：http://www.ics.uci.edu/~lopes/datasets/

(注4)：http://camstudio.org/

```
package task2;

import java.io.File;
import java.io.IOException;

public class Task2 {

    /**
     * 入力として与えられるファイルオブジェクトの中身をバイト配列に読み込み，
     * そのバイト配列を出力する．
     * 出力される配列はファイルにあるバイト列の長さと同じ大きさである．
     * (つまり，配列に余りがない状態で出力される)
     *
     * また，このメソッドは以下の仕様を満たすものとする．
     * 1.ファイルの中身が無い場合は，長さ0の配列を返す．
     * 2.ファイルが存在しない場合は，nullを返す．
     *
     * @param file
     * @return
     * @throws IOException
     */
    public byte[] task2(File file) throws IOException {
        //TODO
    }
}
```

図 7　被験者に与えられたタスク T2 のソースファイル
Fig. 7　Source file of task T2, which was passed to research participants.

表 1　書き忘れが発生したタスク(注5)
Table 1　Tasks and participants where missing middle code occurred(注6).

	T1	T2	T3	T4	T5	T6	合計
P1	–	√		√		√	3
P2	√	√	√	√	√	√	6
P3	–	√	√	√		√	4
P4	√	√	√	√	√	√	6
P5	–	√	√	√		√	4
P6	√	√	√	√	√	√	6
P7	√	√	√	√	√	√	6
P8	√	√	√	√			4
P9				√		√	2
合計	5	8	7	9	4	8	41

4.2　被　験　者

本実験には九名の被験者に参加していただいた．1 名は博士研究員，2 名は博士課程の大学院生，6 名は修士課程の大学院生である．全ての被験者は，最低でも半年の Java によるプログラミングの経験がある．被験者の Java プログラミングの経験は，大学の演習や自身の研究を遂行することによって得られた．

4.3　タ　ス　ク

各被験者には六つのタスクが与えられた．被験者はタスクに記載されている仕様を満たすように Java のメソッドを実装した．全てのタスクは単一の Java メソッドを実装するタスクであった．各タスクにおいて，実装するメソッドのシグネチャと Javadoc コメントに記載された仕様が被験者に与えられた．図 7 は，被験者に与えられたタスクの例を表す．また，被験者に与えられたタスクの一覧を以下に示す．

T1　与えられたファイルパスから拡張子を除いたファイル名を取得する．

T2　与えられたファイルの中身をバイト配列に書き込む．

T3　与えられた長方形の左上と右下の座標を取得する．

T4　与えられたバイト配列の一部をフィールドにあるバイト配列で置き換える．

T5　第一引数で与えられたファイルの中身を第二引数で与えられたファイルにコピーする．

T6　与えられた複数の文字列配列（char 型の配列）を連結し，CSV 形式で単一の文字列（String 型）に変換する．

(注5)：T はタスク，P は被験者を表す．
(注6)：T and P represent tasks and research participants, respectively.

表 2　書き忘れたコードの種別と発生したタスクの数
Table 2　Categories of missing middle code and the number of tasks where such code occurred.

書き忘れたコードの種別	発生したタスクの数
引数に対するエラー処理	25
オブジェクトの生成や初期化	19
処理結果に依存した処理の分岐	16
return 文を先に記述する習慣	5
入出力オブジェクトに対する close 処理	3
その他	12

4.4　調査項目 1 への回答

表 1 は，どの被験者のどのタスクにおいて書き忘れが発生したのかを表している．チェック（“√”）は，その被験者のそのタスクにおいて書き忘れが発生したことを表している．ハイフン（“–”）は，そのタスクにおいて動画の録画が失敗したため，書き忘れが発生したかどうかの調査ができなかったことを表す．全 54 タスクのうち 3 タスクにおいて録画が失敗していた．

表 1 から，約 80%のタスク（51 タスク中の 41 タスク）において書き忘れが発生したことがわかる．また，全ての被験者は少なくとも二つのタスクにおいて書き忘れを発生させており，4 人の被験者は全てのタスクにおいて書き忘れを発生させていた．

著者らは，どのようなコードが書き忘れられていたのかについて調査を行った．表 2 はその結果をまとめている．一つのタスクにおいて複数の書き忘れが発生していた場合はそれらを個別に計上している．もっとも多かったのは，メソッドの引数が正常な値を保持しているかをチェックするコードであった．このようなコードの書き忘れは 25 タスクで発生しており，書き忘れたコードが発生したタスクの 61%を占めていた．

表 3　適切な補完候補が存在していたタスク(注7)
Table 3　On which participant's tasks the tool was able to suggest appropriate candidates(注8).

	T1	T2	T3	T4	T5	T6	合計
P1	-		○	○	○	●○	4
P2	●○		●○	●○	○	●○	5
P3	-				○	●	2
P4	○	○		●○		●○	4
P5	-	○				○	2
P6	○					○	2
P7	●○	●○	○	○	○	○	6
P8	○					○	2
P9	○	○	○	○		○	5
合計	6	4	4	5	4	9	32

オブジェクトの生成や初期化のし忘れ，呼び出したメソッドの返り値による処理の分岐漏れも発生していた．五つのタスクでは，return 文を先に書いたことにより，書き忘れと判定されたコードも存在していた．この場合は，return 文を先に記述することにより，Eclipse のエディタ上のコンパイルエラーを取り除くという意図があったと思われる．既存のコード補完手法を利用した場合には，return 文を先に書いてしまった場合は，その上部にコードを補完できないが，書き忘れたコードを考慮した補完手法では上部のコードを補完できる．よって書き忘れたコードを考慮した場合にのみ補完できるという状況という意味では，この場合も計上してよい．

以上のことから，調査項目 1 への回答は，約 80%のタスクにおいて書き忘れが発生した，となる．

4.5　調査項目 2 への回答

表 3 は，どのタスクにおいて適切な補完候補が存在していたのかを示している．なお，構文的には完全に同じではなくても，意味的に同じプログラム文を提示できた場合には，提示できたとして計上した．構文的には同じではないが意味的には同じであった例としては，“`if(i - 1 > j)`” がスーパークローンに含まれる補完候補であり，完成形のコードは “`if(i > j + 1)`” であったというものが挙げられる．表において，白丸（“○”）は書きかけのコードに続く適切な補完候補が存在していたことを表し，黒丸 (“●”) は書き忘れたコードに対する適切な補完候補が存在していたことを表す．

この表より，約 63%のタスク（全 51 タスクのうち 32 タスク）において，適切な補完候補が存在していた．また，書き忘れが発生していた 41 タスクのうちの 10 タスクにおいて，書き忘れたコードに対する適切な補完候補が存在していた．また，この表より，各被験者について少なくとも二つのタスクにおいて，適切な補完候補が存在していたことがわかる．特に，被験者 P7 については，全てのタスクにおいて適切な補完候補が存在していた．また，タスク T6 については，全ての被験者について適切な補完候補が存在していた．

(注7)：白丸（“○”）は書きかけのコードに続く補完候補を表し，黒丸（“●”）は書き忘れたコードに対する補完候補を表す．
(注8)：White circles (“○”) and black circles (“●”) represent appropriate following code and middle code were suggested, respectively.

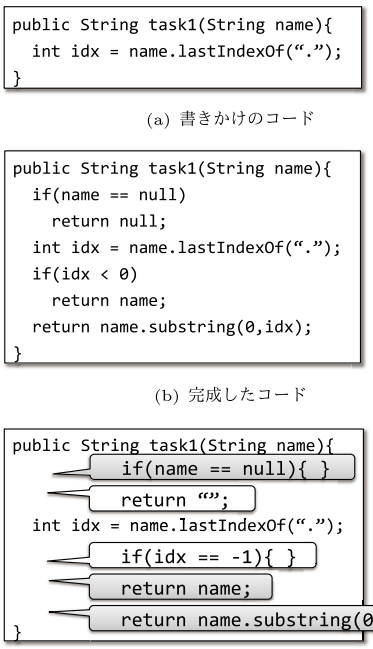


図 8　適切なコードを提示できた例
Fig. 8　Example of tasks where appropriate code was suggested.

図 8 は書き忘れたコードに対する適切な補完候補の例を表している．図 8 (a) は被験者が一つの変数宣言文を書いた状態を表している．また，図 8 (b) はその被験者の完成形のコードを表している．図 8 (c) は図 8 (a) の状態のコードに対する補完候補を表している．図 8 (b) と図 8 (c) を比べると，完成形のコードに含まれるプログラム文のうちの三つがスーパークローンに含まれていた．それら三つの補完候補は図 8 (c) では色付けして示されている．

幾つかのタスクにおいては適切な補完候補が存在していなかった．この原因は，被験者がデータセットに含まれているコードとは別の方法でタスクを実装した

ためである．そのような場合でも完成したプログラムは正しく動作していた．

以上より，調査項目 2 への回答は，書き忘れが発生した 41 のタスクのうち，10 のタスクにおいて適切な補完候補が存在していた，となる．

5. 議　　論

この章では，既存コードのプログラム文を再利用することでコード補完を行うことについて議論する．また，実験の妥当性についても述べる．

5.1 コード補完を用いる利点について

コードの補完を行うことにより，実装を行う際にさまざまな利点がある．第一の利点としてキータイプ数が減ることが挙げられる．キータイプ数を減らすことに特化したコード補完手法も存在する [4]．

第二の利点として，開発者が具体的なコードをイメージできていない場合でも支援できることが挙げられる．既存プログラムから抽出されたコードが開発者に提示されるため，そのコードを閲覧することにより，実装したい機能がどのようなコードになるのかを把握できる．

第三の利点として，コードの品質確保につながることが挙げられる．例えば，開発者がテストを行って失敗した場合に，コード補完機能を利用してテストを成功させるのに必要なコードを補完できれば，コードの品質がある程度確保されたことになる．

5.2 補完候補の再利用によりコードクローンが発生する点について

補完候補を再利用した場合に，補完候補元メソッドと補完対象メソッドにおいて共通のプログラム文が発生することになる．つまり，ソフトウェア間のコードクローンが増加してしまう．コードクローンの存在は，ソフトウェアの保守を難しくする要因の一つといわれており，実際にコードクローンがソフトウェアの保守作業を妨げた事例が報告されている [17]～[19]．その一方で，保守に悪影響を与えるコードクローンは全体の一部のみであるとも報告されている [20]～[22]．

悪影響を与えるコードクローンを開発者のソフトウェアに作りこまないためには，データベースを構築するために用いるソースコードが重要である．もし，データベースの構築に利用されたソースコードがよくテストされたコードや長年の運用実績があるコードである場合は，開発者のソフトウェアに挿入されるコードはそのような信頼性の高いコードである．コードクローンがソフトウェアの保守に悪影響を与えている理由は，ソースコードの異なる複数箇所を同時に変更する必要があるからである．変更が必要な箇所を見落としてしまったり，誤った変更を加えたりした場合には，一貫性がないコードが発生してしまう [17]．

以上のことから，補完候補の再利用によりソフトウェア間のコードクローンを増加させてしまうが，補完候補データベースの構築に信頼性のあるソースコードを利用すれば，補完されたコードがそのソフトウェアの保守作業を妨げてしまう可能性は低い，と著者らは考えている．

5.3 実験の妥当性について

ここでは，本論文の実験の妥当性について述べる．

自動判定方法　本論文で紹介した自動判定手法では，再利用可能なプログラム文を含む全てのメソッドをスーパークローンとして検出できているわけではない．検出できない理由として二つの理由がある．

一点目は，自動判定手法ではシグネチャ文からもハッシュ値を生成しスーパークローンの検出に利用していることである．そのため，再利用可能なプログラム文がシグネチャの異なるメソッド内に存在している場合には，そのメソッドはスーパークローンとして検出されない．しかしシグネチャ文も考慮することで，メソッドのシグネチャのみを書いた状態で補完候補を探すことができるという利点もある．

二点目は，メソッド内のプログラム文の並び順を過度に考慮している場合があることである．**3.2** で述べたように，スーパークローンの検出の際にはプログラム文の並び順を用いてフィルタリングを行っているが，並び順が違っていてもスーパークローンとして検出すべき場合がある．例えば，“*int index = 0;*” と “*float ratio = 0f;*” のように，データ依存が存在しないプログラム文については並び順が違っていてもスーパークローンとして検出しても問題ない．プログラム文間のデータ依存を考慮することでより適切にスーパークローンを検出することは可能である．本論文では，検出の高速性を意識したためにデータ依存の情報は利用しなかった．

以上の二点が解決された自動判定法を用いた場合は，本実験の結果よりも適切な補完候補が存在していたタスク数が恐らく増える．

被験者　実験に参加した被験者は著者らと同じ研究室に所属する博士研究員と大学院生である．Java プログラミングの経験がより豊かなプログラマが被験者

だった場合は実験の結果が変わるかもしれない．本実験で確認できたことは，開発者が Java プログラミングの経験が半年以上程度であった場合は，適切な補完候補が存在していた，ということである．

タスク　実験で用いたタスクはどれも規模が小さく，一つのメソッドとして実装するものであった．一つのタスクが複数のメソッドやクラスを実装することにより完成するような，より実用的なものであった場合は，実験の結果が異なる可能性はある．しかし，一つのタスクの規模が大きくなった場合でも，メソッドを実装していくということには変わりはない．よってより大きなタスクの場合でも，今回の実験と同様に多くの場合において適切な補完候補を提示できると著者らは考えている．

この実験を行う際に著者らがデータセットに対して事前に調査を行い，各タスクについて再利用可能なメソッドが少なくとも 1 種類(注9)は存在していることを確認した．ここで再利用可能なメソッドとは，そのコードの一部若しくは全部を挿入するとタスクを完成させることができるメソッドを意味する．また，実験の手順 3 を行った際に，全てのタスクに対して再利用可能なメソッドが少なくとも 2 種類は存在しており，タスクによっては 10 種類以上のメソッドが存在していたことも確認できた．

書き忘れたコードが発生するのかどうかは被験者の実装に依存する．本実験では，著者らがタスクを作成した後に，被験者がそのタスクを Java メソッドとして実装した．そのため，書き忘れたコードが発生するタスクを著者らが意図的に作成したということはないことがいえる．

データセット　実験では UCI データセットを利用して補完候補データベースを構築した．異なるソースコードを利用してデータベースを構築した場合には，提示される補完候補が異なるため，実験の結果が変わる．また，**5. 2** では，信頼性のあるコードを利用してデータベースを構築することが重要であると述べたが，この実験では補完候補を提示できるかどうかについてのみ調査している．この実験では補完されたコードの信頼性については扱っていない．

(注9)：ここで種類とは，コードが同じメソッドを表す．例えば，3 種類のメソッドとは，コードが異なる（クローンになっていない）メソッドが三つはあるという意味である．

6. 関連研究

Hill らはメソッド単位でのコード補完手法を提案した [5]．この手法では，書きかけのメソッドを入力として受け取り，そのメソッドから特徴ベクトルを生成する．そして，あらかじめ収集しておいたメソッド群から生成した特徴ベクトルと比較し，最も近い特徴ベクトルをもつメソッドを，その書きかけのメソッドの完成形であるとして提示する．

山本らは，書きかけのコードに続くコードを補完する手法を提案した [10]．この手法では，収集したソースコードに含まれる各メソッドを字句列に変形する．そして，各字句の区切りにおいて字句列を二分割し，前半の字句列をキー，後半の字句列をバリューとして，そのペアをデータベースに格納する．開発者が書きかけのコードに対してコード補完機能の呼び出しを行った場合は，その書きかけのコードがキー字句列としてデータベースに問い合わせが行われる．そして，同じキー字句列をもつペアがデータベースに存在した場合は，そのペアのバリュー字句列が書きかけのコードに続くコードとして補完される．

Han らは，開発者が短い入力を与えると，それをコンパイル可能なコードへと展開する手法を提案している [4]．例えば，開発者が “*pv st nm*” と入力すると，この手法はそれを “*private String name*” と変換する．このように，この提案手法を利用することで大幅にキータイプ数を減らすことができる．

また，メソッド呼び出しの補完手法も存在する．Bruch らはコード例を学習することによって書きかけのメソッド呼び出しを補完する手法を提案した [2]．この手法では，三つの補完戦略を利用している．一つ目の戦略は，収集したコードにおいて多く利用されているメソッド呼び出しが補完の候補として提示する．二つ目の戦略は，収集したコードにおけるメソッド呼び出しの並びの順を学習することにより，書きかけのコードにおいて次にどのメソッド呼び出しが必要かを予測する．三つ目の戦略は，k 近傍法を用いて，収集したコードの中から書きかけのコードと似ているコードを特定し，その特定したコードにおいて呼び出されているメソッド呼び出しを利用する．

Omar はコードの補完を支援するシステム Graphite を開発した [8]．Graphite は色名と正規表現のコード補完を支援する際に，コードそのものを提示するのではなく，より開発者にわかりやすい形での補完を支援

する．色名の場合は，色名を出すだけではなく，その色を実際に表示し，開発者が選べるようにしている．正規表現については，正規表現内に存在する構文誤りの指摘機能や開発者が正規表現の振る舞いを調査するためのテストケース作成を支援する．

API の呼び出しパターンを抽出することによるコードの補完支援も幾つか提案されている [3], [7], [12]．ライブラリの API には決まった呼び出しパターン（同時に使う API や API 呼び出しの順序）があることから，既存コードから API がどのように呼び出されているのかを抽出し，その情報を用いて書きかけのコードに対して補完を行う手法である．API 呼び出しの情報を用いていることから，API 呼び出しを含むプログラム文やそれらの文で利用されている変数を利用している他のプログラム要素が補完の対象となる．

上記以外にもコードの補完手法は提案されているが [6], [9], [11]，これらは字句単位のコードの補完であり，プログラムの文単位で補完を行う手法ではない．

統合開発環境には，実装を効率的に行うための支援機能が装備されている．例えば，Eclipse には Quick Fix という機能がある．Quick Fix には，さまざまなプログラム要素に対して，幾つかの変更方法が定義されている．例えば例外を投げるメソッドの呼び出しを含むプログラム文が try-catch 文で囲まれていない場合にこの機能を使うと，そのプログラム文は try-catch 部分で囲まれる．また，参照しているフィールドが未定義であった場合にこの機能を使うと，その未定義フィールドを作成する．このように Quick Fix は，ツールがさまざまな変更方法を定義するという支援方法である．一方，本研究で扱うコード補完は，データから補完候補を抽出する．そのため，データを用意できればコード補完を行えるため，新しい変更方法の対応のためにツールを拡張する必要はない．

本論文における判定方法は，Type-3 クローンの検出手法を含んでいる．Type-3 クローンの検出手法はこれまでにも多く研究がされている．有名なものとしては，Roy らが開発したクローン検出ツール Nicad が挙げられる [23]．このツールは，ソースコードに含まれる各ブロック（メソッドやその内部の for 文等）について，最長共通部分列を特定することによってクローンを検出する．しきい値以上類似しているブロックのペアがクローンとして検出される．しかしながらこの手法は，大規模ソースコードからの検出には長い時間を必要とする．また，この手法では，コードの補完には利用できないようなメソッド（例えば，図 5 (c)，5 (d)，5 (e) のメソッド）までクローンとして検出してしまう．そのため，著者らはスーパークローンの検出において，既存の手法を利用しなかった．

7. むすび

本論文では，書き忘れたコードの発生頻度（調査項目 1）と，書き忘れたコードを考慮することによるコード補完機会の向上（調査項目 2）について調査を行った．この調査では，九名の被験者が Java のコードを実装した．実装の様子は動画で撮影され，その動画を著者らが閲覧することにより，調査項目 1 を調査した．その結果，書き忘れは約 80%のタスク（51 タスク中の 41 タスク）において発生しており，それらのうちの 24%のタスク（41 タスク中の 10 タスク）において書き忘れたコードに対する適切な補完候補が存在していたことを示した．

以上のことから書き忘れたコードを補完対象とすることでコード補完を行える機会が向上する．本論文で述べた調査手法は，書きかけのメソッドを与えると，補完候補を含んだメソッドを自動的に検出する．そのため，書き忘れたコードを対象とするコード補完手法の一部として利用することができる．しかし，コード補完手法とするためには，複数の補完候補が存在した場合のフィルタリングや順位付けが必要である．今後はこれらについて研究を進めていきたい．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号：25220003)，挑戦的萌芽研究 (課題番号：24650011)，及び文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の助成を得て行われた．

文　献

[1] G.C. Murphy, M. Kersten, and L. Findlater, “How are Java software developers using the eclipse IDE?,” IEEE Softw., vol.23, no.4, pp.76–83, 2006.

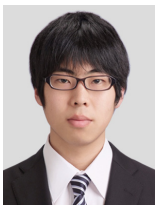
[2] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.213–222, 2009.

[3] R.J. Akbar, T. Omori, and K. Maruyama, “Mining API usage patterns by applying method categorization to improve code completion,” IEICE Trans. Inf. & Syst., vol.E97-D, no.5, pp.1069–1083, May 2014.

[4] S. Han, D.R. Wallace, and R.C. Miller, “Code completion from abbreviated input,” Proc. 2009

IEEE/ACM International Conference on Automated Software Engineering, pp.332–343, 2009.
[5] R. Hill and J. Rideout, “Automatic method completion,” Proc. 19th International Conference on Automated Software Engineering, pp.228–235, 2004.
[6] D. Hou and D.M. Pletcher, “Towards a better code completion system by API grouping, filtering, and popularity-based ranking,” Proc. 2nd International Workshop on Recommendation Systems for Software Engineering, pp.26–30, 2010.
[7] 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. 34th International Conference on Software Engineering, pp.69–79, 2012.
[8] C. Omar, Y. Yoon, T.D. LaToza, and B.A. Myers, “Active code completion,” Proc. 34th International Conference on Software Engineering, pp.859–869, 2012.
[9] R. Robbes and M. Lanza, “Improving code completion with program history,” Automated Software Engineering, vol.17, no.2, pp.181–212, 2010.
[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,” 情処学論，vol.53, no.2, pp.644–652, 2012.
[11] C. Zhang, J. Yang, Y. Zhang, J. Fan, J. Zhao, and P. Ou, “Automatic parameter recommendation for practical API usage,” Proc. 34th International Conference on Software Engineering, pp.826–836, 2012.
[12] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “MAPO: Mining and recommending API usage patterns,” Proc. 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, pp.318–343, 2009.
[13] T. Ishihara, Y. Higo, and S. Kusumoto, “How often is necessary code missing? — A controlled experiment,” Proc. 14th International Conference on Software Reuse, pp.156–163, 2015.
[14] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Does return null matter?,” Proc. International Conference on Software Maintenance, Reengineering and Reverse Engineering, pp.244–253, 2014.
[15] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” IEEE Trans. Softw. Eng., vol.33, no.9, pp.577–591, 2007.
[16] B.S. Baker, “On finding duplication and near-duplication in large software systems,” Proc. 2nd Working Conference on Reverse Engineering, pp.86–95, 1995.
[17] 肥後芳樹，楠本真二，“コード修正履歴情報を用いた修正漏れの自動検出,” 情処学論，vol.54, no.5, pp.1686–1696, 2012.
[18] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, “Experience of finding inconsistently-changed bugs in code clones of mobile software,” Proc. 6th International Workshop on Software Clones, pp.94–95, 2012.
[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code,” IEEE Trans. Softw. Eng., vol.32, no.3, pp.176–192, 2006.
[20] N. Göde and R. Koschke, “Frequency and risks of changes to clones,” Proc. 33rd International Conference on Software Engineering, pp.311–320, 2011.
[21] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software,” Proc. Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pp.73–82, 2010.
[22] C.J. Kapser and M.W. Godfrey, “‘cloning considered harmful’ considered harmful: Patterns of cloning in software,” Empirical Software Engineering, vol.13, no.6, pp.645–692, 2008.
[23] C.K. Roy and J.R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” Proc. 16th International Conference on Program Comprehension, pp.172–181, 2008.

（平成 27 年 9 月 4 日受付，11 月 24 日再受付，
12 月 29 日早期公開）



石原　知也

平成 24 年大阪大学基礎工学部情報科学科卒業．平成 26 年同大学大学院博士前期課程修了．在学時，コードクローン分析やコード補完に関する研究に従事．

肥後　芳樹　（正員）

平成 14 年大阪大学基礎工学部情報科学科中退．平 18 同大学大学院博士後期課程修了，平成 19 年同大学情報科学研究科助教．平成 27 年同准教授．博士 (情報科学)．ソースコード分析，特にコードクローン分析やリポジトリマイニング，リファクタリング支援に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．

楠本　真二　（正員）

昭和 63 大阪大学基礎工学部卒業．平成 3 年同代学大学院博士課程中退．同年同大額基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 同大学大学院情報科学研究科助教授．平成 17 年同教授．博士 (工学)．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．情報処理学会，IEEE，JFPUG 各会員．