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

信学技報
TECHNICAL REPORT OF IEICE.

コードクローンを用いたコード補完手法の提案

石原　知也†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科

あらまし　ソースコードの実装を効率化する手段の 1 つにコード補完がある．コード補完は開発者が途中まで実装したコードの不足部分を補完する機能であり，近年ではソースコードの再利用支援を目的としたコード補完手法も提案されている．しかし，既存のコード補完手法は途中まで実装されたソースコードの後に続く部分のみを補完するにとどまり，開発者が実装途中に書き漏らしたコードを補完することはできない．本研究では，書き漏らしたコードの発生状況を調べ，書き漏らしコードの補完が有効であるかを調査する．また，書き漏らしコードの補完も実現するために，コードクローンを用いたコード補完手法を提案して調査に用いる．提案する手法は途中まで実装されたソースコードの Type-3 コードクローンを見つけることで，そのソースコードの後に続く部分だけでなく書き漏らしたコードを特定し補完することが可能になる．本論文では，被験者実験を行いコードの書き漏らしについて調査を行った．調査の結果，多くの被験者の実装において書き漏らしが発生しており，書き漏らしコードの補完は有益であることが確認された．
キーワード　コードクローン，コード補完，ソースコード再利用

Code Completion with Code Clones

Tomoya ISHIHARA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University

Abstract　Code completion is one of the techniques that raise efficiency of implementation. Code completion techniques make up for code lacking in half-written code. Recently, some researchers have proposed techniques using code completion for helping code reuse. However, existing code completion techniques, which complete only code following the half-written code, cannot complete code in the middle of the half-written code. In this research, we investigate the situation where developers forget to write some code and examine whether it is useful to complete the middle code. Also, we propose a new code completion technique with code clone detection techniques in order to complete the middle code. The proposed technique can complete both the middle code and following code by detecting Type-3 code clones of the half-written method. In this research, we have conducted an experiment with subjects to investigate the code that subjects forgot to write. As a result, we have confirmed that most of the subjects forgot to write some code in their given tasks and that it was useful to complete the middle code.
Key words　Code Clone, Code Completion, Source Code Reuse

1. はじめに

コードの実装を効率化させる手段の 1 つにコード補完がある．コード補完とは，実装途中のコードを入力として与えると，これから先に実装されるであろうコードを自動的に補完する機能のことを示す．開発者は必要な機能を途中まで実装しコード補完を適用することで，必要な機能の実装を完了させることができ，開発者の実装コストの削減が期待できる．現在では，いくつかの IDE [1,2] が変数名やメソッド呼び出し名などの字句を補完する機能を有しており，開発者は簡単にコード補完を使用することができる．実際，多くの開発者が IDE のコード補完を頻繁に使用している [3]．IDE が行う補完機能は単純なものであり，多くの研究者はより効率的な補完手法を提案している [4,5]．

また，近年では，コードの再利用支援のためにコード補完を使用するという研究も行われている [6–8]．書きかけのコードに対して既存研究を用いたコード補完を適用すると，書きかけのコードに続く文が補完される．しかし，既存のコード補完はすでに存在する書きかけのコードの途中や前にコードを補完することは考慮していない．開発者が途中まで実装したコードに実装漏れがあることは十分に考えられることである．そのため，既存手法を用いたコード補完は開発者が書き漏らしたコードに

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


4 }
```

(a) 補完の対象となるメソッド

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

(b) 既存手法を適用した際の補完候補

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

(c) 書き漏らしを考慮した際の補完候補

図 1　既存手法による補完とその問題点

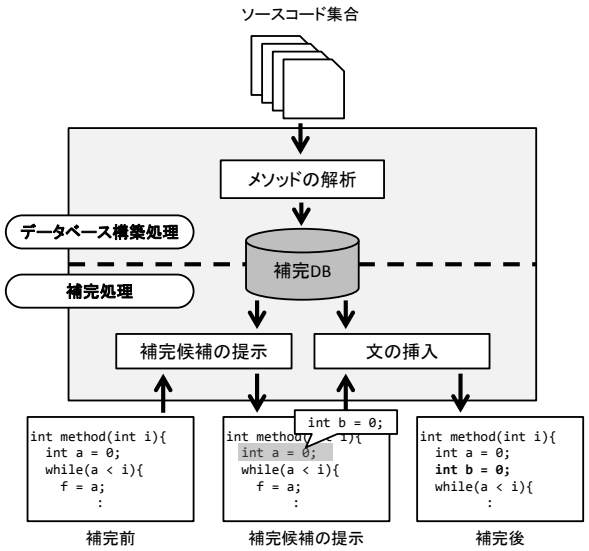


図 2　提案手法の概要

対するコードの再利用の機会を失っている可能性がある．

本論文では，書き漏らしコードの発生状況について調査を行い，実装の際にどの程度書き漏らしが発生しているかを明らかにする．開発者が実装の際に書き漏らしを多く発生させている場合は，その分だけ既存手法を用いたコード補完でコードの再利用の機会が失われている．そのため，書き漏らしたコードも補完できるような補完手法が必要になる．また，書き漏らしコードを補完することでコードの再利用が促進されるかも調査する．本論文では，コードクローンを用いた新しいコード補完手法を提案し，書き漏らしコードを補完する．提案する手法は Type-3 コードクローンを検出し，検出されたコードクローンから補完候補となる文を特定する．それに伴い，本論文では高速な Type-3 コードクローン検出手法を考案して，補完候補の特定に用いている．

本論文では被験者実験を行い，書き漏らしコードが発生する状況や補完される状況を調査した．実験では，被験者はいくつかのタスクが与えられ，被験者がタスクを実装する様子が録画された．記録された映像を基に，書き漏らしの発生状況や提案手法の適用結果を著者らが調査した．調査の結果，被験者が実装したメソッドの 80%で実装途中に書き漏らしが発生していることがわかった．また，提案手法は被験者の実装したメソッドの 63%に対して有益な補完候補を提示しており，そのうち 31%は書き漏らしたコードに対するものであった．

2. 研究の動機

2.1 既 存 研 究

山本らは，大量のコードを解析して補完用コーパスを作成し，そのコーパスの情報を基に適切な補完コードを検索して補完する手法を提案している [9]．補完用コーパスにはメソッドを字句の列に変換したものが含まれている．この字句列は任意の地点で 2 つに分割され，分割された字句列のうち先頭のものが key，後方のものが value として保存されている．開発者はメソッドの冒頭部を記述して山本らの手法を適用する．この時，山本らの手法では入力された冒頭部を字句列に変換し，得られた字句列とマッチする key を検索する．マッチする key があれば，対応する value を補完する．これによって，開発者はメソッドの冒頭のみを記述するだけでその後に続くロジックを実装することができる．

2.2 既存研究の課題点

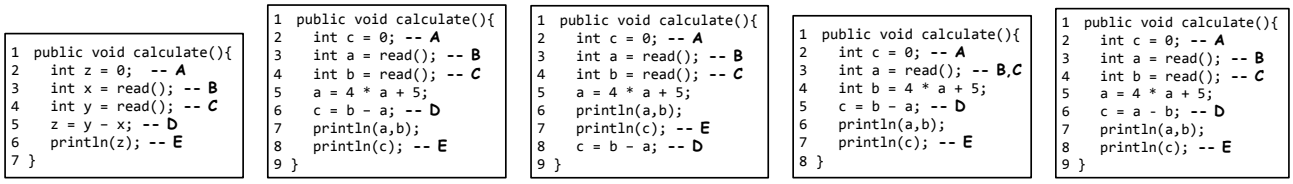
図 1(a) は実装途中のメソッドである．開発者は引数で与えられた長方形の左上の座標と右下の座標を取得して，それを定められた形式で出力するメソッドを作成している．図 1(a) のメソッドは左上の座標の取得まで実装されている．この時，開発者が図 1(a) のメソッドに対してコード補完機能を適用すると，右下の座標を取得し定められた形式に変換した上で出力する処理が補完されることが期待される．

開発者が図 1(a) のメソッドに対して既存手法を用いたコード補完を適用した場合，図 1(b) のように補完候補が提示される．しかし，図 1(c) のように開発者はメソッドの先頭で引数に対するエラーチェックを記述し忘れている可能性があったが，その部分は補完候補として提示されていない．既存手法では，途中まで実装されたコードを冒頭部に持つメソッドを基に，そのコードの後に続く部分を補完候補として提示する．そのため，開発者が途中まで実装したコードの前や間に補完候補が提示されることはない．このように，既存手法では書き漏らしコードを補完することは考慮されていない．よって本論文では，既存手法で補完することのできない書き漏らしコードがどの程度発生するかについて調査する．また，書き漏らしコードを補完することができれば，開発者はより信頼性の高いコードを効率的に実装できると考えられる．そこで本論文では，書き漏らしコードを補完することで開発者が得られる効果についても調査する．

以上より，本論文では以下に示す 2 つの問題を調査する．

RQ1： 実装中に書き漏らしはどの程度発生するか．

RQ2： 書き漏らしコードを補完することでコードの再利用はどの程度促進されるか．



(a) 入力メソッド　(b) 入力メソッドのコードクローン　(c) 文の順番が異なるメソッド　(d) 文の対応関係に重複があるメソッド　(e) 変数の対応関係が一貫していないメソッド

図 3　コードクローンとして検出されるメソッドと除外されるメソッドの例

3. 書き漏らしコードに対するコード補完

3.1 提案手法の概要

本論文では，書き漏らしコードに対する新しいコード補完手法を提案し，調査に用いる．提案する手法はコードクローン検出技術を利用する．この手法では，入力として与えられたメソッドの Type-3 コードクローンとなるメソッドを検出し，そのメソッドを基に補完候補となる文を検出して開発者に提示する．

提案手法の概要を図 2 に示す．提案手法はデータベースを構築する処理と補完処理の大きく 2 つの処理に分けることができる．また，補完処理は補完候補の提示処理と文の挿入処理の 2 つに分けることができる．提案手法は与えられたソースコードからメソッドを抽出し，抽出したメソッドを解析して得た情報をデータベースに登録しておく．開発者は補完したいメソッドに対して提案手法を適用すると，提案手法はデータベースの情報を基に補完候補となる文を求め，補完箇所をハイライトする形で補完候補を提示する．開発者は提示された補完候補をみてそのコードを補完するかどうかを判断する．補完したい場合は補完候補を選択することで補完が実行される．開発者は必要な数だけ補完候補を選択したら補完を終了させる．

図 2 を見てわかるように，提案手法は大きく分けて以下の 3 つの処理に分類できる．

メソッドの解析 この処理では，与えられたソースコードを解析して補完用のデータベースを作成する．データベースには，ソースコードから抽出されたメソッドの文の情報が格納される．ここでいう文の情報には，文中で使われている変数名や文のハッシュ値が含まれる．文の情報は次の処理である補完候補の提示の際に使用される．

補完候補の提示 この処理では，入力として与えられたメソッドの Type-3 コードクローンとなるメソッドを検出し，補完候補となる文を特定する．Type-3 コードクローンの検出には後述する Type-3 コードクローン検出法を用いる．入力されたメソッドと検出された Type-3 コードクローンの差分となる文が補完候補となる．特定された補完候補となる文は各補完箇所ごとに開発者に提示される．

文の挿入 この処理では，選択された補完候補となる文を実装中のコードに補完する．開発者が複数の補完候補を選択することも考慮して，補完後も補完候補の提示は継続する．開発者がコードに対して補完以外の処理を行ったら，補完候補の提示を終了する．

補完候補の提示処理を実行するためには，それ以前にメソッドの解析処理が必ず実行されていなければならない．提案手法はデータベースの情報を基に補完候補となる文を特定しているため，あらかじめメソッドの解析処理を実行してデータベースを構築する必要がある．メソッドの解析処理は 1 度だけ実行されればよい．一方で，補完候補の提示処理は開発者がコード補完機能を利用するたびに実行される．同様に，文の補完処理も提示された補完候補が選択されるたびに実行される．

3.2 Type-3 コードクローンの検出

提案手法は，開発者が入力したメソッドに対してデータベースから Type-3 コードクローンを検出する．そのため，開発者の実装の速度を落とさないためにも，Type-3 コードクローンの検出は高速である必要がある．しかし，既存の Type-3 コードクローン検出では大量のコードに対するコードクローン検出に時間を要するため，本論文で用いる Type-3 コードクローン検出には適さない．そのため，本論文では新しい高速な Type-3 コードクローン検出法を考案し，提案手法で使用する．

考案した Type-3 コードクローン検出法は入力としてメソッドを受け取り，入力されたメソッドの Type-3 コードクローンとなるメソッドを出力する．ただし，Type-3 コードクローン検出法を用いるためにはあらかじめソースコードを解析してメソッドの情報をデータベースに保存しておく必要がある．データベースには，解析したソースコードに存在するメソッドとそのメソッド中にある文の情報が格納されている．文の情報には文のハッシュ値と文中で使用される変数名が含まれる．データベースに情報が格納されたら，Type-3 コードクローン検出法を用いてコードクローン検出を行うことができる．

Type-3 コードクローン検出法は以下の 3 つのステップで構成される．

STEP1: 入力されたメソッドの各文のハッシュ値を算出し，全てのハッシュ値を持つメソッドをデータベースから検索する．検索されたメソッドが Type-3 コードクローンの候補になる．また，検索時に入力メソッドと検索されたメソッドに対して文の対応関係を構築する．ハッシュ値が等しい文同士が対応関係をもつ．

STEP2: 検索されたメソッドの対応関係をもつ文と入力メソッドの対応関係をもつ文の順番が異なる場合は，検索されたメソッドを Type-3 コードクローンの候補から除外する．また，重複して対応関係をもつ文がある場合も検索されたメソッドを Type-3 コードクローンの候補から除外する．

STEP3: 対応関係を持つ文から得られる変数の対応関係が一貫しているかを確認する．一貫していない場合はそのメソッ

ドを Type-3 コードクローンの候補から除外する．最後まで候補として残ったメソッドが Type-3 コードクローンとして検出される．

図 3 はコードクローンとして検出されるメソッドとそうでないメソッドの例を示している．図 3 の各メソッドにおいて，文の後ろにあるアルファベットが等しい文同士が対応関係をもつ．図 3(a) のメソッドが入力されたとき，図 3(b) はコードクローンとして検出される．一方で，図 3(c), 3(d), 3(e) のメソッドはコードクローンではない．図 3(a) のメソッドの 5, 6 行目はそれぞれ図 3(c) のメソッドの 8, 7 行目に対応する．しかし，それぞれのメソッドで文の順番が異なる．図 3(a) メソッドの 3, 4 行目はどちらも図 3(e) のメソッドの 3 行目に対応しており，重複がある．これらのメソッドは STEP2 でコードクローンの候補から除外される．

また，図 3(a) のメソッドの 3, 4 行目とそれに対応する図 3(e) のメソッドの 3, 4 行目から変数 x と a，y と b が対応していると判断される．しかし，図 3(a) のメソッドの 5 行目とそれに対応する図 3(e) のメソッドの 6 行目から変数 x と b，y と a が対応していると判断される．結果として，対応関係をもつから得られる変数の対応関係が一貫していないため，STEP3 でコードクローンの候補から除外される．

4. 実　　験

本論文では，提案手法を *Eclipse Plugin* として実装し実験を行った．また，実験の際に提案手法に与えるソースコードとして *UCI source code data set*(以降 *UCI dataset* と表記する) を用いる [10]．*UCI dataset* は web 上で公開されている Java ソフトウェアを集めて構成された非常に大規模な Java のソースファイル集合である．表 1 は *UCI dataset* の構成を示している．

4.1 定　　義

本論文では，実装途中のコードの後に続くコードを**未実装コード**，実装途中のコードの前や間に入るコードを**書き漏らしコード**と呼ぶ．また，本実験では，書き漏らしコードをすでに実装されている文の前や間に実装される文と定義する．ただし，未完成の文の前に実装された文は書き漏らしコードではない．

4.2 実 験 方 法

本論文で行った実験は大きく以下の 2 つの STEP に分けられる．

STEP1: 被験者にいくつかのタスクを与え，そのタスクを被験者が実装する．実装の様子は録画される．

STEP2: 記録した映像を基に著者が 2.2 で述べた 2 つの RQ を調査する．

STEP1 では，被験者は与えられた仕様を満たすメソッドを実装する．被験者は録画環境が整っているコンピュータに接続して実装を行う．実装の際は被験者に対する制限はなく，被験者は普段実装する時と同じ環境で実装できる．例えば，web でコードを検索してそのコードをコピーして実装することも許可されている．あらかじめ各メソッドに対していくつかのテストケースが用意されており，その全てを通過したときにタスクが終了したと判断される．

表 1　*UCI source code data set* の構成

プロジェクト数	13,193
Java ファイル数	2,127,877
メソッド数	20,449,896

STEP2 では，STEP1 で記録した映像を著者が見て 2 つの RQ を調査する．RQ1 の調査では，被験者がタスクを実装する様子を映像を通じて確認し，書き漏らしが発生したメソッドやその状況を記録する．書き漏らしコードの発生状況は文単位で調査される．

RQ2 の調査では，被験者がタスクを実装する流れを映像から取得し，その流れに沿って途中まで実装したコードに対して提案手法を適用する．本論文で実装したツールは入力したメソッドの構文が正しくなければならないという制限がある．そのため，実装途中でメソッドの構文が正しくなるたびに提案手法を適用し，補完候補が提示されるかを確認する．

4.3 被　験　者

本実験の被験者は著者と同じ専攻に所属する 8 名と研究員 1 名の計 9 名である．すべての被験者は Java の経験が半年以上ある．また，被験者の Java の使用目的として，1 人を除いて全員が授業でのプログラミングと研究での使用を挙げている．

4.4 タ　ス　ク

本実験で被験者に与えられるタスクの数は 6 であり，その全てが与えられた仕様を満たすメソッドを実装するというものである．被験者には宣言部 (修飾子，返り値，名前，引数，throws 節) だけが記述されたメソッドが与えられ，コメント部分にはそのメソッドの満たすべき仕様が記述されている．被験者はそのメソッドの本文を実装する．各タスクにはテストケースが用意されており，全てのテストケースを通過したらタスクが終了したと判断する．また，最初に全てのテストケースを通過した際の実装をその被験者の最終実装とみなし，それ以後のコードの追加，変更，削除は考慮しない．表 2 は本実験で使用したタスクの一覧を示している．

4.5 RQ1：書き漏らしはどの程度発生するか

RQ1 の調査結果を表 3 に示す．表 3 は被験者が各タスクを実装中に書き漏らしコードが発生したかを示している．表中の「○」はタスクの実装中に 1 度でも書き漏らしコードが発生した

表 2　実験で使用したタスク

タスク	概要
タスク 1	与えられたパスから拡張子を除いたファイル名を取得する．
タスク 2	与えられたファイルの中身をバイト配列に読み込む．
タスク 3	与えられた長方形の左上と右下の座標を取得する．
タスク 4	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．
タスク 5	与えられた 2 つファイルのうち，片方のファイルの中身をもう片方にコピーする．
タスク 6	与えられた文字列配列を csv 形式で 1 つの文字列に連結する．

ことを表し，空白は書き漏らしなく実装が終わったことを表す．また，表中の「-」は録画が失敗したタスクを示す．

表 3 の結果から，録画が成功した全 51 のタスクのうち 41 のタスクで書き漏らしコードが発生したことが判明した．これは全タスクの約 80%に相当する．また，タスクや被験者ごとに多少のばらつきはあるものの，どのタスクや被験者においても 1 度は書き漏らしコードが発生していることもわかった．このことから，開発者は日常的に書き漏らしコードを発生させている可能性が高いと考えられる．

加えて，書き漏らしコードが発生する原因についても調査を行った．表 4 は書き漏らしコードの発生原因とそのコードが発生したタスクの合計数を示している．1 つのタスクに複数の書き漏らしコードが有りそれらの発生原因が異なる場合は，それぞれ別々に数えられている．表 4 の結果から，最も多い書き漏らしコードの発生原因は「引数に対するエラー処理の漏れ」であることがわかった．その数は 25 であり，書き漏らしコードが発生した全タスクの約 61%を占める．比較的多い発生原因として「オブジェクトの初期化忘れ」と「処理結果に対する分岐忘れ」があることもわかった．また，書き漏らしコードの発生原因として「return 文を先に記述」が挙げられている．これは 4.1 で述べた書き漏らしコードの定義によって引き起こされたもので，必ずしも書き漏らしに該当するとは限らない．しかし，既存研究を用いたコード補完を適用する場合，return 文を削除する等の前処理が必要となる可能性がある．

以上の結果から，RQ1 に対する回答は実装したメソッドの約 80%で書き漏らしが発生するとなる．

4.6 RQ2：書き漏らしコードを補完することでコードの再利用はどの程度促進されるか

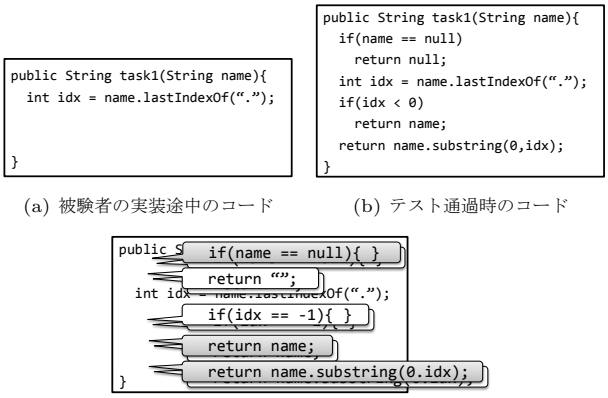
RQ2 の調査結果を表 5 に示す．表 5 は各タスクごとに被験者が実装した文と意味的に等しい文が提示されたかを示す．ここでいう被験者が実装した文と意味的に等しい文とは，被験者が実際に実装した文かまたは被験者が実際に実装した文と構造的には異なるが意味的には等しい文のことを示す．被験者が実際に実装した文と意味的に等しい文とは，例えば，ツールの提示した文が `if(i - 1 > j)` で被験者が実際に実装した文が `if(i > j + 1)` といった場合を示す．表中の「○」は被験者が実装した文と意味的に等しい文が 1 つでも提示されたことを示す．このうち，書き漏らしコードに対する提示があった場合は「●」で表される．

表 3　書き漏らしコードが存在したタスク

	タスク 1	タスク 2	タスク 3	タスク 4	タスク 5	タスク 6	計
被験者 A	-	○		○		○	3
被験者 B	○	○	○	○	○	○	6
被験者 C	-	○	○	○		○	4
被験者 D	○	○	○	○	○	○	6
被験者 E	-	○	○	○		○	4
被験者 F	○	○	○	○	○	○	6
被験者 G	○	○	○	○	○	○	6
被験者 H	○	○	○	○			4
被験者 I				○		○	2
計	5	8	7	9	4	8	41

表 4　書き漏らしコードの発生原因

発生原因	該当するタスク
引数に対するエラー処理の漏れ	25
オブジェクトの初期化忘れ	19
処理結果に対する分岐忘れ	16
return 文を先に記述	5
close 処理の漏れ	3
その他	12



(c) ツールによる補完候補となる文の提示

図 4　有益な補完の例

表 5 から，32 のタスクで被験者が実装した文と意味的に等しい文が提示されていることがわかる．これは全タスクの 63%を占める．また，書き漏らしコードの補完に絞ると，書き漏らしコードの存在したタスクの 24%にあたる 10 のタスクで被験者が実装した文と意味的に等しい文が提示されていることがわかる．書き漏らしコードが補完されたタスクは補完が行われたタスクの 31%に相当する．

表 5 を見ると，被験者ごとにばらつきはあるがどの被験者の実装に対しても有益な補完候補が提示されている．一方で，タスクに注目して表 5 を見ると，タスク 1 とタスク 6 はすべての被験者の実装に対して有益な補完候補が提示されている．また，書き漏らしコードの補完にのみ着目すると，特定の被験者やタスクにおいて有益な補完候補が提示されている．

1 回の補完でいくつの文が補完されるかについても調査した．書き漏らしコードが補完される場合，その際の文の数は全ての場合で 1 または 2 であった．一方で，未実装コードの場合は 2 文以上補完されることが多かった．また，補完候補が提示される状況についても調査し，補完適用時に実装されている文が少ないほど補完候補が多く存在することを確認した．

表 5　各タスクごとの補完結果

	タスク 1	タスク 2	タスク 3	タスク 4	タスク 5	タスク 6	計
被験者 A	-		○	○	○	●	4
被験者 B	●		●	●	○	●	5
被験者 C	-				○	●	2
被験者 D	○	○		○		●	4
被験者 E	-	○				○	2
被験者 F	○					○	2
被験者 G	●	●	○	○	○	○	6
被験者 H	○					○	2
被験者 I	○	○	○	○		○	5
計	6	4	4	5	4	9	32

図 4 はある被験者が途中まで実装したコードに対して書き漏らしコードと未実装コードの両方が補完された例である．図 4(a) のメソッドは被験者がタスクを 1 文だけ実装した時のメソッドの状態を示しており，テスト通過時には図 4(b) の実装に至った．一方で，図 4(a) のメソッドに補完を適用すると，図 4(c) のように補完候補となる文が提示される．図 4(b) と 4(c) を比較すると，図 4(c) の黒背景で表示されている補完候補となる文を被験者は実際に実装していることがわかる．

以上の結果から，RQ2 に対する回答は実装したメソッドの約 30%で書き漏らしコードに対するコードの再利用が行われるとなる．

5. 議　　論

5.1 コードクローンを生成することの是非

提案手法は入力されたメソッドの Type-3 コードクローンとなるメソッド (以降，補完候補メソッドと呼ぶ) を基に補完候補となる文を提示している．そのため，補完によって生成されたメソッドは補完候補メソッドのコードクローンになりうる．コードクローンの存在はソフトウェアの保守性に悪影響を及ぼすといわれており，補完よってコードクローンの悪影響が伝搬する可能性がある．しかし，近年の研究では全てのコードクローンがソフトウェアに悪影響を及ぼすわけではないということがわかっている [11, 12]．また，補完候補メソッドとなるのはデータベースに存在するメソッドなので，提案手法に与えるソースコードを成熟したソフトウェアから構築すればコードクローンの悪影響の伝搬を抑えることができる．

5.2 妥当性への脅威

本論文の結果の妥当性について，以下に挙げる点に留意する必要がある．

被験者 本論文の被験者は全員が著者と同じ専攻に属している．そのため，別の大学の学生や産業界での経験が豊富な開発者に対して同じ実験を行うと結果が変わる可能性がある．

タスク 本論文で用いたタスクはどれも小規模で汎用的な処理を実装するものである．実験結果からもわかるように，補完候補が提示されるかどうかはタスクに依存する可能性が高い．そのため，今回用いたタスクを変更することで本論文の結果と異なる結果が得られる可能性がある．

データセット 提案手法は入力されたメソッドの Type-3 コードクローンをデータベースから検出することで補完を可能にしている．そのため，補完候補の有無はデータベースを構成する際に与えるソースコードに依存する．本論文の実験では，Java のオープンソースソフトウェアで構成されるデータセットを用いているが，与えるソースコードの規模や構成を変えることによって結果が変わる．

6. おわりに

本論文では，開発者が書き漏らしたコードの発生状況や書き漏らしコードを補完することによる開発者への効果を調査した．また，調査に際して，コードクローンを利用した新しいコード補完手法を提案して書き漏らしコードの補完を実現した．本論文では，2 つの RQ を調査するために被験者実験を行った．実験の準備として，被験者にタスクを与え，そのタスクを被験者が実装する流れが録画された．録画した映像を基に，書き漏らしコードが発生する状況や補完される状況を調査した．その結果，調査対象のメソッドの約 80%で書き漏らしがあり，コードの書き漏らしは頻繁に発生していることが明らかになった．また，調査対象のメソッドの約 60%で提案手法は有効な補完候補を提示しており，そのうち約 30%は書き漏らしコードに対するものであった．

今後の課題として，書き漏らしコードの発生状況について更に詳しく調査を行うことを考えている．本論文で行った実験をより多くの被験者とタスクで行うことによって，開発者やタスクと書き漏らしコードの関係がより明確になると考えられる．

謝辞　本論文は，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)，および文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の支援を受けた．

文　　献

[1] “Eclipse”. `http://www.eclipse.org`.
[2] “Intellisense”. http://blogs.msdn.com/b/vcblog/archive/tags/intellisense/.
[3] G.C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the eclipse ide?,” IEEE Software, vol.23, no.4, pp.76–83, 2006.
[4] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” Proc of The 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.213–222, 2009.
[5] C. Omar, Y. Yoon, and T.D.L. adnB. A. Myers, “Active code completion,” Proc. of the 34th International Conference on Software Engineering, pp.859–869, 2012.
[6] 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. of the 34th International Conference on Software Engineering, pp.69–79, 2012.
[7] R. Hill and J. Rideout, “Automatic method completion,” Proc. of the 19th International Conference on Automated Software Engineering, pp.228–235, 2004.
[8] R. Holmes and R.J. Walker, “Systematizing pragmatic software reuse,” ACM Transactions on Software Engineering and Methodology, vol.21, no.4, pp.1–44, 2012.
[9] T. Yamamoto, N. Yoshida, and Y. Higo, “Seamless code reuse using source code corpus,” Proc. of the 5th International Workshop on Empirical Software Engineering in Practice, pp.31–36, 2013.
[10] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, “Uci source code data sets”. `http://www.ics.uci.edu/~lopes/datasets/`.
[11] K. Hotta, Y. Sasaki, Y. Sano, Y. Higo, and S. Kusumoto, “An empirical study on the impact of duplicate code,” Advances in Software Engineering, vol.2012, no.5, 2012.
[12] N. Göde and R. Koschke, “Frequency and risks of changes to clones,” Proc. of the 33th International Conference on Software Engineering, pp.311–320, 2011.