修士学位論文

題目

コードクローンを用いたコード補完

指導教員
楠本 真二 教授

報告者
石原 知也

平成 26 年 2 月 5 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 25 年度 修士学位論文

コードクローンを用いたコード補完

石原 知也

内容梗概

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，実装途中のコードの不足部分を自動的に補う機能のことを示し，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．しかし，既存のコード補完手法は実装途中のコードに欠落が無いことを前提とするため，欠落した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り，このようなコードも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．

本研究では，実装途中のコードの後に続く部分の補完だけでなく書き漏らしの可能性があるコードも補完候補として提示する手法を提案する．提案手法は，補完候補となるメソッドを Type-3 クローンの一種を用いて特定している．それに伴い，本研究では高速な Type-3 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．

本研究では提案手法を *Eclipse plugin* として実装して被験者実験を行い，書き漏らしの発生状況と提案手法の有効性を調査した．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%は書き漏らしコードに対するものであったことも判明した．

主な用語

コードクローン
コード補完
ソースコード再利用
書き漏らしコード

目 次

図 目 次

表 目 次

1　まえがき

コードの実装を効率化させる手法の 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 [1–3] がトークン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 [4]．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 [5, 6].

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 [7–9]．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，開発者が実装を忘れていたコードも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実装できる．このように，補完されるべきコードとして開発者が途中まで実装したコードの後に続くコードと開発者が実装漏れしたコードの 2 種類が考えられるため，これら両方を補完することができるコード補完が必要である．

また，既存のコード補完機能では補完候補となる文をまとめて 1 つの補完候補として提示する．しかし，補完されたコードの中には開発者が必要としないコードが含まれている可能性がある．あくまで開発者は補完候補として最も適切なコードを提示されたコードの中から選択しただけであり，それが開発者が必要としているコードと完全に一致するとは限らない．結果として，既存のコード補完機能を使用した場合，1 度補完されたコードを削除して書き直すことが必要になる可能性がある．そのため，開発者が補完に適さないと判断したコードを補完候補から効率的に除外することが望まれる．

本研究では，実装途中のコードの後に続く部分の補完だけでなく書き漏らしの可能性があるコードも補完候補として提示し，開発者が補完候補となる文を自由に選択することで開発者が必要とするコードのみを補完する手法を提案する．提案手法は，補完候補となるメソッドを Type-3 クローンの一種を用いて特定している．それに伴い，本研究では高速な Type-3 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．提案手法は，補完される位置ごとに補完候補となる文を提示する．そのため，開発者が補完しなくてもよいと思った箇所は補完候補を選択しなければ良い．これによって，1 度補完されたコードを開発者が書き直す必要がなくなる．また，既存のコード補完機能とは異なり，提案手法を使うことで，開発者は補完箇所ごとに異なるメソッドから求められた文を補完することもできる．

本研究では提案手法を *Eclipse plugin* として実装して被験者実験を行った．実験では，書き漏らしの発生状況と提案手法の有効性を調査した．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%は書き漏らしコードに対するものであったことも判明した．

以降 2 章では，コード補完やコードクローンの定義について述べる．3 章では研究動機について述べる．4 章では提案手法について説明し，5 章で本研究での実装方法について述べる．6 章では評価実験の方法とその結果について述べ，7 章でその考察や妥当性の検証を行う．8 章では関連研究について述べ，最後に 9 章で本研究のまとめと今後の課題について述べる．

2　準備

2.1　コード補完

コード補完とは，開発者が途中まで書いたコードを入力として与えると不足している部分を推測してそのコードを自動的に補完する機能のことを示す．多くのコード補完は，補完候補を開発者に提示する処理と開発者が選択した補完候補を挿入する処理の 2 つから構成される．開発者が途中まで記述したコードに対してコード補完を適用すると，いくつかの補完候補が提示される．提示された候補の中から開発者が 1 つを選択することによって，その候補が編集中のコードに挿入される．コード補完機能を使うことによって，開発者は必要なコードを全て手動で実装する必要がなくなるため，実装効率が向上する．また，手動によるコードの記述ミスを軽減する効果もあり，これも実装効率を高める 1 つの要因となる．

現在までに，多くの研究者がコード補完手法を提案している [5–12]．これらのコード補完手法を実装したツールも多数開発されており，いくつかの IDE を通じてその機能を利用することができる．本論文では，コード補完手法を以下の 2 種類に分類する．

字句の補完

この補完では，途中まで記述された文を基に次に続くであろう変数やメソッド呼び出しなどの字句を補完する．例えば，Eclipse におけるコードアシスト機能がこれに該当する．

文の補完

この補完では，途中まで記述されたコードを基に不足している文を補完する．字句の補完とは異なり，この補完は過去に実装されたコードを効率的に再利用することで実装効率の向上を図っている．既存研究として，API 呼び出しとその周辺でよく使われるコードをまとめて補完する手法が提案されている [7]．

2.2　コードクローン

2.2.1　定義

コードクローンとはソースコード中に存在する同一，あるいは類似するコード片のことである．図 1 に示すように，ソースコード中に存在する 2 つのコード片 α，β が類似しているとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるコード片も類似していないとき，α，β を極大クローンと呼ぶ．また，互いにクローンであるコード片を同値としたときの同値類をクローンセットと呼ぶ [13]．

ただし，どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる．

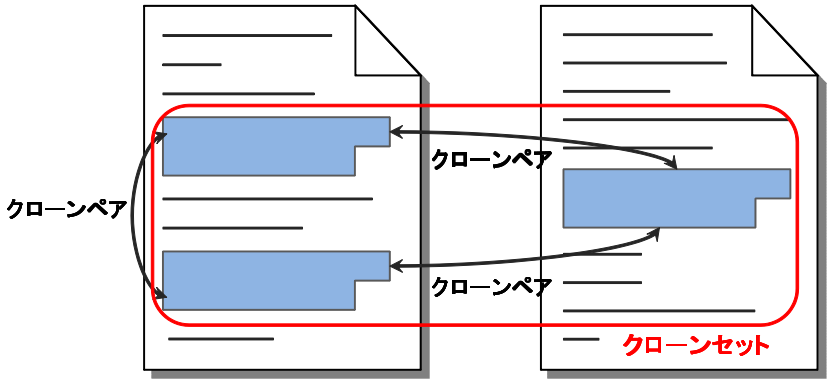


図 1: クローンペアとクローンセット

また，コードクローン間の類似の度合いに基づきコードクローンを次の 3 つのタイプに分類することができる [14] [15].

Type-1

空白やタブの有無，括弧の位置などのコーディングスタイルに依存する箇所を除いて，完全に一致するコードクローン.

Type-2

変数名や関数名などのユーザ定義名，また変数の型など一部の予約語のみが異なるコードクローン.

Type-3

Type-2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン.

2.2.2 発生の原因

コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものが挙げられる [16] [17] [18].

既存コードのコピーアンドペーストによる再利用

近年のソフトウェア設計手法を利用することにより構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーアンドペーストによる場当たり的な既存コードの再利用が多く行われるようになった．コピーアンドペーストによって生成されたコード片は，コピー元のコード片とコードクローン関係になる．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである．

適切な機能の欠如

抽象データ型やローカル変数を用いることができないプログラミング言語を開発に用いている場合，同じようなアルゴリズムを用いた処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるソフトウェアにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し記述することによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールによって生成されるコードは，あらかじめ決められたコードをベースにして自動的に生成される．このため，類似した処理を目的としたコードを生成した場合，識別子名などの違いを除き，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは，各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い．

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

(a) 対象メソッド

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

(b) 対象メソッドのスーパークローン

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

(c) 文の順番が異なるメソッド

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

(d) 扱うオブジェクトの整合性がとれないメソッド

図 2: スーパークローンとなるメソッドとならないメソッドの例

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.2.3 メソッドに対するスーパークローン

本研究では，補完候補を特定するためにメソッド単位の Type-3 コードクローンの一種を用いる．このコードクローンを本論文では**スーパークローン**と呼ぶ．スーパークローンは 2 つのメソッドに対する単方向の関係を表し，メソッド M_1 がメソッド M_2 のスーパークローンであるとき M_2 は M_1 のスーパークローンではない．あるメソッド M_1, M_2 について，M_1 が M_2 のスーパークローンであるための必要十分条件は以下の 3 つである．ここで，S_1, S_2 はそれぞれ M_1, M_2 にある文の集合を表し，$S_1(i)$ は M_1 の i 番目の文のことを表す．

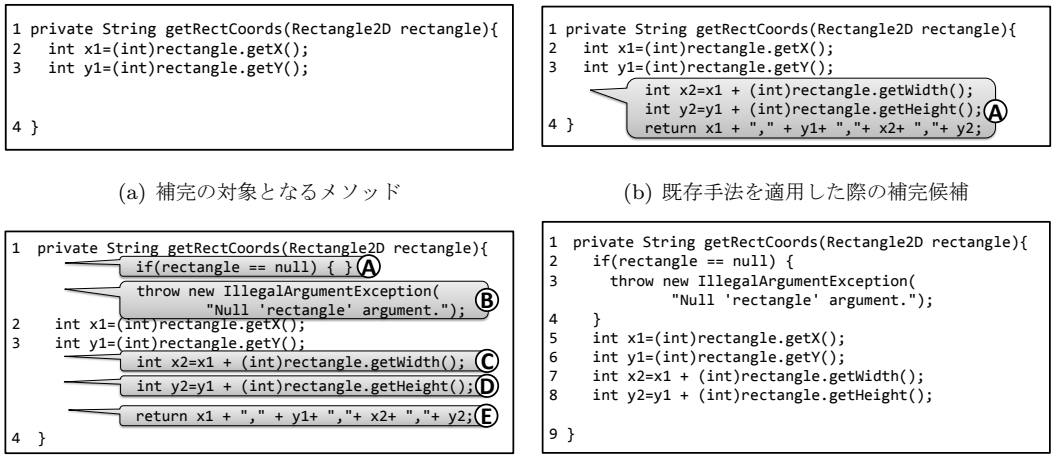
1. M_1 に存在する任意の文 $S_1(i)$ に対して，$S_1(i) = S_2(j)$ となる j が存在する．ただし，$S_1(i) = S_2(j)$ ならば，i を除く任意の整数 k に対して $S_1(k) \neq S_2(j)$ が成り立つ．ここで，$S_1(i) = S_2(j)$ の関係を $S_1(i)$ と $S_2(j)$ が対応していると呼び，$S_1(i)$ と $S_2(j)$ が同じ文であることを示す．

2. $S_1(m) = S_2(s)$ かつ $S_1(n) = S_2(t)$ のとき，$m < n$ ならば $s < t$ が成り立つ．

3. M_1 に存在する任意の文 $S_1(i)$ と，$S_1(i)$ と対応関係をもつ $S_2(j)$ に対して，$S_1(i)$ と $S_2(j)$ から得られる変数の対応関係に一貫性がある．ここで，変数の対応関係とは対応関係をもつ 2 つの文の同じ位置で使われている変数を結びつけたものである．例えば，`x=y+z` と `a=b+c` という対応関係をもつ文がある場合，x と a，y と b，z と c という変数の対応関係が得られる．

図 2 はスーパークローンと判断されるメソッドとそうではないメソッドの例を示している．図 2 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もの同士が対応関係をもつ．一方で，赤字で表示されている文は対応関係を持たない．図 2(a) のメソッドに対して，図 2(b) のメソッドはスーパークローンになっている．一方で，図 2(c) のメソッドは図 2(a) のメソッドのスーパークローンではない．図 2(a) のメソッドの 5, 6 行目はそれぞれ 2(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において文の順序関係が異なっている．また，図 2(d) のメソッドも図 2(a) のメソッドのスーパークローンではない．図 2(a) のメソッドの 3, 4 行目とそれに対応する図 2(d) 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 2(a) のメソッドの 5 行目とそれに対応する図 2(d) 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，スーパークローンではないと判断される．

メソッド M_2 がメソッド M_1 のスーパークローンである場合，M_1 に対して適切な文を加えるだけで M_2 を作成することができる．そのため，スーパークローンを補完候補の特定に用いた場合，M_2 と M_1 の差分となる文が補完候補として特定される．



(a) 補完の対象となるメソッド

(b) 既存手法を適用した際の補完候補

(c) 提案手法を適用した際の補完候補

(d) 提案手法が提示した補完候補のうち **A**,**B**,**C**,**D** を選択した場合

図 3: 既存手法と提案手法の違い

3 研究の動機

3.1 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 [19]．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 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．

3.2 既存研究の課題点

図 3(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ようとしている．図 3(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 3(a) のメソッドに対してコード補完機能を利用すると，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 3(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 3(b) のように補完候補が提示される．既存手法では途中まで実装されたコードの後に続くコードしか補完できないため，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．また，既存手法では補完候補となるすべての文をまとめて 1 つの補完候補として提示する．つまり，3(b) に

おいて，補完候補として **A** を選択すると **A** にあるすべての文が 1 度に補完されることになる．そのため，提示された補完候補の中に必要のない文が含まれている場合でもその分が補完されてしまう．これによって，開発者は不要な文を削除しなければならなくなる．

一方，図 3(c) は，図 3(a) のメソッドに書き漏らしコードがあることを考慮にいれた場合に提示される補完候補を示している．開発者は図 3(a) のメソッドを実装中に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，その部分も補完候補として提示されている．また，提示されている補完候補は 1 つの文のみで構成されており，開発者は提示されたそれぞれの文に対して補完を行うかを選択することができる．例えば，図 3(c) の補完候補のうち **A**,**B**,**C**,**D** を選択すると，図 3(d) のメソッドが得られる．このような提示方法を採用することで，開発者は必要な文のみを補完させることができる．

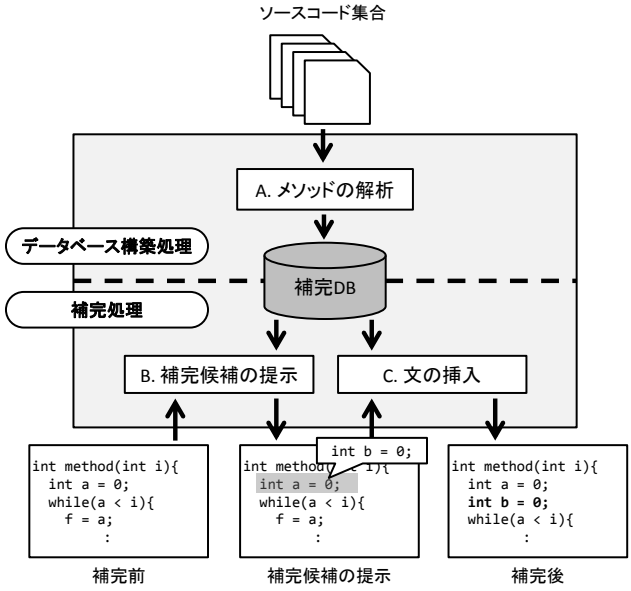


図 4: 提案手法の概要

4 提案手法

本研究では，コードクローンを利用したコード補完手法を提案する．この手法では，入力として与えられたメソッドのスーパー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．提案手法は補完候補として特定された文を 1 つの文ごとに提示する．これによって，開発者は必要な文のみを補完することが可能になる．

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スーパー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，スーパー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 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，本研究で用いるスーパークローン検出には適さない．そのため，本研究では新しい高速な Type-3 クローン検出法を考案し，提案手法内で使用する．

図 4 は提案手法全体の構成を示している．図 4 を見てわかるように，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．

4.1 定義

本研究では，コード補完が適用されるメソッドを**補完対象メソッド**，コード補完を適用した結果開発者に提示される文を**補完候補文**，補完候補文を特定する基となるメソッドを**補完候補メソッド**と呼ぶ．また，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．図 3(a) のメソッドに対して補完を適用する場合，提案手法は図

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

図 5: 補完対象となるメソッドのスーパークローンとなるメソッド

5 のようなメソッドをデータベースから検索し補完候補となる文を特定する．このとき，図 3(a) のメソッドが補完対象メソッド，図 5 のメソッドが補完候補メソッドとなる．補完候補メソッドから特定され，図 3(c) で提示されている各文が補完候補文となる．補完候補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．また，補完候補文は補完候補メソッドと補完対象メソッドの差分となる文である．

4.2 スーパークローンの検出

考案した Type-3 クローン検出法 (以降スーパークローン検出法と呼ぶ) 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スーパークローンとなるメソッドを出力する．ただし，スーパー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集合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集合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，スーパークローン検出法を用いてクローン検出を行うことができる．

スーパー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スーパー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スーパー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スーパークローンの候補から除外する．これによって，スーパークローンの条件 1,2 を満たすメソッドのみがスーパークローンの候補として残る．

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スーパー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スーパークローンとして検出される．

4.3 提案手法の概要

4.2 を踏まえた，提案手法の概要を図 4 に示す．提案手法は与えられたソースコード集合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文を求め，補完箇所をハイライトする形で補完候補文を提示する．開発者は提示された補完候補文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文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文を選択したら補完を終了させる．

図 4 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集合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，ソースコード集合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スーパークローンとなるメソッドを検出し，補完候補文を特定する．入力されたメソッドと検出されたスーパークローンの差分となる文が補完候補文となる．特定された補完候補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文を選択することも考慮して，補完後も補完候補文の提示は継続する．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データベースを構築するために 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文が選択されるたびに実行される．

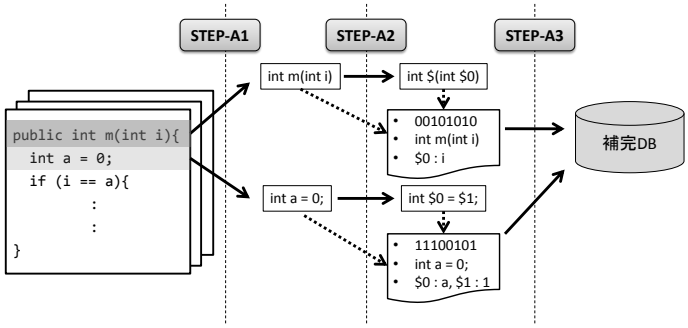


図 6: メソッド解析処理の概要

5 実装

この章では，4 章で説明した提案手法の各処理の実装について説明する．本研究で実装したツールは，現在のところ Java 言語のみを対象とする．

5.1 メソッドの解析

提案手法はあらかじめ与えられたソースコード集合からメソッドを抽出して解析を行う．抽出されたメソッドは文に分割され，各文の情報を取得する．得られた情報はデータベースに保存される．提案手法は以下のステップを通じてメソッドを抽出し，解析を行う．

STEP-A1: ソースコード集合からメソッドを抽出し，文に分割する．

STEP-A2: 各文に対して変数名等の正規化を行う．

STEP-A3: 各文をハッシュ化し，得られた情報とともにデータベースに保存する．

図 6 はメソッド解析処理の流れを示している．以降各ステップについて詳しく説明する．

5.1.1 STEP-A1: 文への分割

このステップでは，まず与えられたソースコード集合からメソッドを抽出する．コードから *Abstract Syntax Tree(AST)* を作ることでメソッドを抽出できる．*AST* の作成には *JavaDevelopmentTools*（*JDT*）[20] を使用している．

次に，抽出したメソッドを文に分割する．文への分割も AST を使用することで簡単に実現できる．分割する際には，インデントや空行などの情報は無視される．分割によって得られた文には，メソッドの先頭から番号が割り振られる．本研究では，すべての文を以下の 3 種類に分類する．

- 単文 (直後にブロックを持たない)
- 複文 (直後にブロックを持つ)
- メソッドのシグネチャ

このとき，Java 言語における文は表 1 のように分類される．単文以外の文は直後にブロックを持つ可能性がある．分割によって得られた文は最終的に開発者に提示される補完候補文となる可能性がある．

5.1.2 STEP-A2: 文の正規化

このステップでは，分割によって得られた各文に対して正規化を行う．正規化とはユーザ定義名を特殊な文字列に置換することである．正規化を行うことによって，ユーザ定義名のみが異なる 2 つの文を同一視することができるようになる．

本研究における正規化の対象は変数名とリテラルである．また，本研究では，各文に対する正規化処理として *Parameterized Matching* を適用する [21]．*Parameterized Matching* は同じ変数に対しては同一の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手法である．つまり，各文において同じ変数が使用されるたびにその変数名が同一の文字列に置換される．通常の正規化ではすべての変数は同じ文字列に置換されるため，*Parameterized Matching* は通常の正規化に比べて正確に文の同一性を判定することができる．例えば，`a = a + b.get(0);` という文に対して *Parameterized Matching* を適用した場合，`$0 = $0 + $1.get($2);` となる．

5.1.3 STEP-A3: 情報の保存

このステップでは，正規化された各文に対して以下の情報を取得し，データベースに保存する．

- 文のハッシュ値
- 文字列としての正規化された文
- 正規化情報．例えば，変数名 a は\$0 に変換された等
- 単文以外の文については，直後のブロックに含まれる最後の文の番号

文のハッシュ値の計算には，128 ビットのハッシュ値を返す *MD5* アルゴリズム [22] を使用する．また，正規化情報とは文中で使用されている変数名と正規化後の名前の対応関係を表す．例えば，変

表 1: Java 言語における文の分類

文の種類	Java 言語における文
単文	セミコロンで終わる文, case, default
複文	do-while, for, if, else, label, try, catch, finally, switch, synchronized, while
シグネチャ	メソッド宣言部

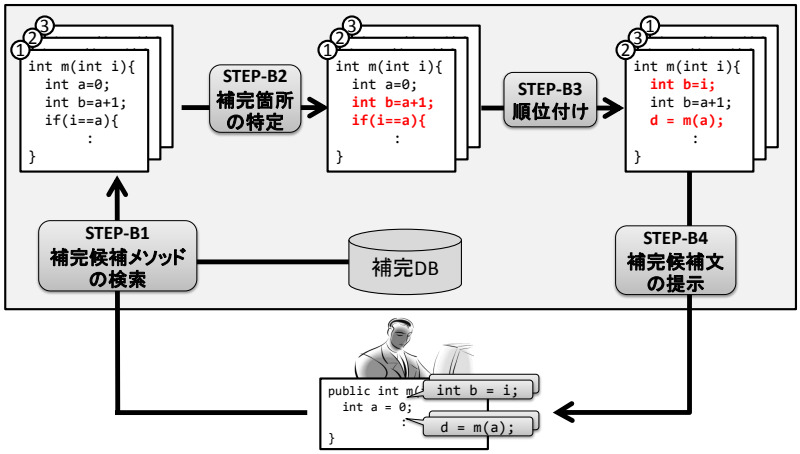


図 7: 補完候補の提示処理の概要

数名 a が\$0 に変換された等である．正規化された文は提示される補完候補文として使用される．正規化情報は補完候補の提示の際に使用される．ブロックに含まれる最後の文の番号は補完箇所の特定に使用される．

5.2 補完候補の提示

開発者がコード補完機能を呼び出すと，提案手法は補完候補の提示処理を実行する．特定された補完候補は補完箇所の直前の文に下線を引くことで開発者に提示される．提案手法はまず補完候補メソッドを特定し，その後に補完候補文を特定する．補完候補メソッドの特定にはスーパークローンを検出する独自のクローン検出手法を使用する．

提案手法は以下のステップを通じて補完候補の提示を行う．

STEP-B1: 入力として与えられたメソッドのスーパークローンをデータベースから検索する．検出されたスーパークローンは補完候補メソッドである．

STEP-B2: 補完候補メソッドから補完候補文を特定し，各補完候補文の補完箇所を求める．

STEP-B3: 補完箇所ごとに補完候補文の提示する順番を決定する．

STEP-B4: STEP-B3 で求めた順番で補完候補文を提示する．

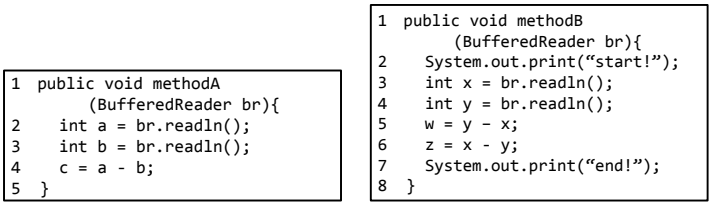
図 7 は補完候補の提示処理の流れを示している．以降各ステップについて詳しく説明する．

5.2.1 STEP-B1: 補完候補メソッドの検出

このステップでは，開発者入力した補完対象メソッドのスーパークローンを検出する．スーパークローンとして検出されたメソッドは補完候補メソッドである．

4.2 の考えを基に，提案手法は以下の流れを通じてスーパークローンを検出する．

Phase-1: 補完対象メソッド T にあるすべての文のハッシュ値を算出する．



(a) 補完対象メソッドのサンプル　(b) サンプルとなるメソッドのスーパークローン

図 8: スーパークローンの関係をもつメソッド

Algorithm 1 Corresponds Statements

Require: $targetMethod, candidateMethod, tIndex, cIndex$

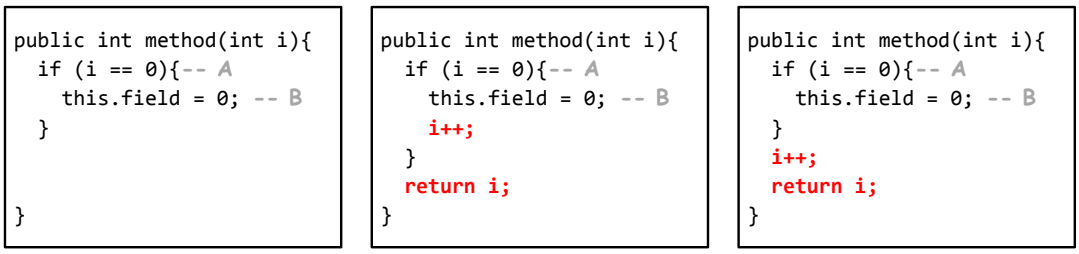
Ensure: $true$ or $false$

```
1: for i = cIndex to candidateMethod.statementLength do
2:   if targetMethod.statement[tIndex] = candidateMethod.statement[i] then
3:     if checkVariable(targetMethod.statement[tIndex], candidatemMethod.statement[i])
       then
4:       if tIndex = targetMethod.statementLength − 1 then
5:         return true
6:       else
7:         if correspondStatement(targetMethod, candidateMethod, tIndex + 1, i + 1) then
8:           return true
9:         end if
10:      end if
11:    end if
12:  end if
13: end for
14: return false
```

Phase-2: T にあるすべての文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は補完候補メソッド群と呼ぶ．

Phase-3: 補完候補メソッド群に属するメソッド C に対して，T の先頭の文と C の先頭の文のハッシュ値を比較する．等しければ，2 つの文に対応関係をもたせる．このとき，T の i 番目の文の対応関係を R_i と表記する．等しくなければ，C を次の文に進めて再度文の比較を行う．これを繰り返し，T のすべての文の対応関係を構築する．T にあるすべての文の対応関係を構築する前に C の最後の文に到達した場合は，C を補完候補メソッド群から除外する．

Phase-4: Phase-3 で構築した全ての R に対して，対応関係を持つ 2 つの文に存在する変数の対



(a) 補完対象メソッド　(b) 補完候補文がブロックの内側にあるスーパークローン　(c) 補完候補文がブロックの外側にあるスーパークローン

図 9: 補完箇所の特定

応関係を構築する．2 つの文の同じ位置で使用されている変数が対応関係をもつ．変数の対応関係は一時的にメモリ上に保存される．構築した変数の対応関係とメモリ上の変数の対応関係が矛盾する場合，Phase-3 で構築した T と C の文の対応関係を変更する．これを矛盾が発生しなくなるまで繰り返す．構築可能な文の対応関係を全て網羅しても矛盾が解消されない場合は，C を補完候補メソッド群から除外する．

Phase-3 と Phase-4 にあたる文の対応関係を構築する処理のアルゴリズムを Algorithm1 に示す．このアルゴリズムは入力として与えられた 2 つのメソッド $targetMethod$ と $candidateMethod$ に対して文の対応関係が構築できるかを判定し，構築できれば真をできなければ偽を返す．また，このアルゴリズムは再帰を用いて繰り返しを実現しており，先頭の文から比較することを示すために引数 $tIndex$ と $cIndex$ に最初に 1 を与える必要がある．Phase-4 まで実行した後に補完候補メソッド群に残っているメソッドがスーパークローンとして検出される．

図 8 を例として補完候補メソッドの検出の流れを説明する．図 8(a) が入力されたメソッドであり，図 8(b) がそのスーパークローンとなるメソッドである．提案手法はまず図 8(a) のメソッドにあるすべての文をハッシュ値に変換し，そのハッシュ値をキーとしてデータベースからスーパークローンの候補を検索する．ここで，図 8(b) のメソッドがスーパークローンの候補として検出される．次に，提案手法は図 8(a),8(b) のメソッドに対して文の対応関係を構築する．先頭からハッシュ値を比較していき，図 8(a) のメソッドの 2,3,4 行目の文と図 8(b) のメソッドの 3,4,5 行目の文が一時的に対応関係を持つ (以降，この対応関係を 2-3,3-4,4-5 と表記する)．このとき，2-3 より変数 a は変数 x と対応関係があるとわかる．しかし，4-5 では変数 b と変数 x に対応関係があり，矛盾が発生する．そのため，手法は文の対応関係を変更する．ここでは，4-5 を 4-6 に変更する．変更後は変数の対応関係に矛盾が発生しないため，提案手法は図 8(b) のメソッドを図 8(a) のメソッドのスーパークローンと判断する．以降，図 8(b) のメソッドは補完候補メソッドとして扱われる．

5.2.2 STEP-B2: 補完箇所の特定

このステップでは，検出された補完候補メソッドから補完候補文を特定し，それぞれの文の補完箇所を求める．ここでいう補完箇所とは，補完対象メソッドを文またはブロックの存在を示す中括弧で区切られた空間のことを示す．ブロックは分岐や繰り返しと連結して制御構造を決定するため，補完箇所がブロックの中か外かでプログラムの挙動が大きく変化する．そのため，補完箇所の区切りの 1 つとしてブロックを表す中括弧を使用している．このステップまでに補完候補メソッドは特定されており，補完候補メソッドにある文の中で補完対象メソッドの文と対応関係を持たない文が補完候補文となる．そのため，それぞれの補完候補文がどこに補完されるかを求める必要がある．

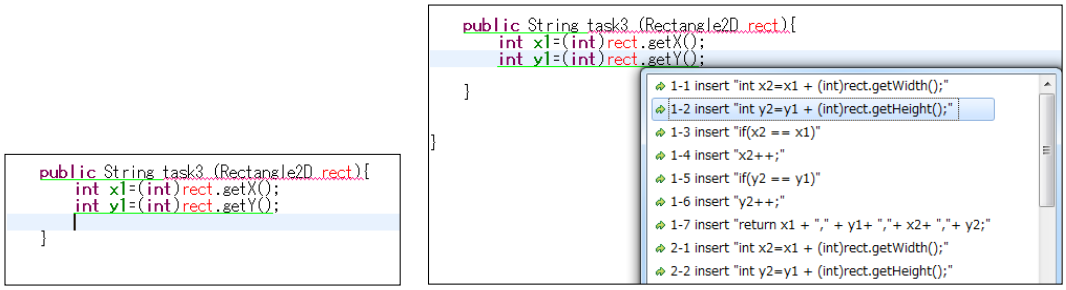
図 9 を用いて，補完箇所特定の流れを説明する．図 9(b),9(c) のメソッドは図 9(a) のメソッドのスーパークローンとして検出されている．また，後ろに記述されているアルファベットが等しい文同士が対応関係を持ち，赤字で記述されている文が補完候補文として特定されている．補完箇所はスーパークローン検出の際に求められた対応関係を基に求められる．補完対象メソッドにある全ての文は補完候補メソッドにある文のいずれかと必ず対応関係をもつ．そのため，補完候補文の直前と直後にある文で対応関係を持つものをみることで，補完対象メソッドのどの文の間に補完すればいいかを判断できる．図 9(b),9(c) のメソッドの場合，補完候補文の直前の文は対応関係 B を持つので，補完対象メソッドの対応関係 B を持つ文の後ろに補完されることがわかる．しかし，このままでは中括弧の情報が抜けているため，データベースに保存されているブロックに含まれる最後の文番号の情報を使用する．これによってブロックの影響範囲が判明するため，補完箇所が決定できる．図 9(b),9(c) のメソッドの場合，2 つのメソッドにある if ブロックの最後の文番号から，それぞれのメソッドにある `i++;` という文を if ブロックの内側に提示するか外側に提示するかを判定できる．

5.2.3 STEP-B3: 順位付け

このステップでは，検出された補完候補メソッドの順位付けを行う．順位が若い補完候補メソッドから特定された補完候補文は提示される順番が早くなる．提示される順番が遅い文ほど，開発者がその文にたどり着くまでにかかる時間が長くなる．そのため，開発者が補完する可能性が高いであろう補完候補文はその提示順番をなるべく早くするほうがよい．本研究では，補完候補メソッドの過去の再利用回数で提示順番を決定している．つまり，過去により多く再利用されたメソッドはその提示順番が早くなっている．過去の再利用回数はデータベース内に存在する同一メソッドの個数を計算することで代用する．この順位付け手法は過去の研究で有効性が示されている [23].

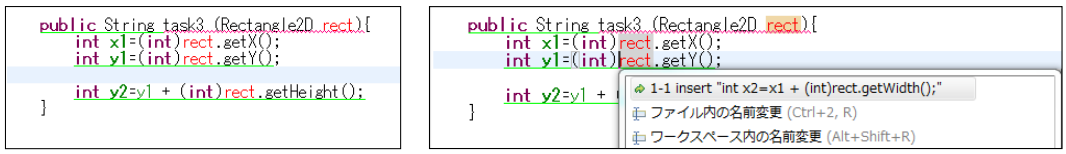
5.2.4 STEP-B4: 補完候補文の提示

このステップでは，特定された補完候補文を開発者に提示する．補完候補文は STEP-B3 で求められた順番で提示される．本研究では，STEP-B2 で求めた補完箇所の直前の文に下線を引くことによって，開発者に補完候補文の存在を明示する．開発者は下線を選択すると，その部分に補完するこ



(a) 補完候補文が提示されているメソッド　　(b) 補完候補文の一覧

図 10: 補完候補文の提示



(a) 補完実行後のメソッド　　(b) 連続した文の補完

図 11: 文の挿入

とができる文の一覧が提示される．開発者は提示された文を見て，どの文を補完するかを決定することになる．図 10 は補完候補文が提示されているメソッドの様子を示している．図 10(a) のメソッドにおいて緑色の下線を選択することで，10(b) のように補完候補文一覧を得ることができる．

5.3　文の挿入

5.2 で提示された文を開発者が選択すると，提案手法は選択された文を補完する．補完候補文がまだ残っている場合は，継続して補完候補文を提示する．開発者がコードを補完以外の形で編集すると，補完が終了する．

提案手法は以下のステップを通じて文の補完処理を行う．

STEP-C1:　選択された文を適切な位置に補完する．

STEP-C2:　補完候補文が残っている場合はそれらの補完箇所を修正する．

以降それぞれのステップについて詳しく説明する．

5.3.1　STEP-C1: 選択された文の補完

このステップでは，開発者が選択した文を適切な位置に補完する．5.2 において，すでに補完候補文の補完箇所は判明している．しかし，実際に補完処理を行う際にはコードのインデントなどを考慮する必要がある．インデントを無視して文を補完してもメソッドの挙動は変わらないが，補完が適用されたメソッドは開発者にとって非常に見づらいものになる．そのため，コードのフォーマットをできるだけ崩さないように補完を行うことが求められる．

本研究では，補完箇所の直後の文のインデントに揃えた形で補完候補文を挿入する．ただし直後の文がブロックの終りを示す中括弧で合った場合は，補完箇所はブロックの中になるためインデントを 1 つずらして挿入する．図 11(a) は図 10(a) のメソッドの補完実行後を表している．図 11(a) を見ると，補完によって生成された文もインデントが統一されていることがわかる．

5.3.2 STEP-C2: 補完箇所の修正

このステップでは，開発者が連続して文の補完行うことを想定して，補完箇所の修正を行う．多くの場合，開発者は 1 つの文のみの補完では満足せず，複数の文に対して補完を行う．この時，直前に選択された補完候補文によって，いくつかの補完候補文の補完箇所を修正しばければならない．例えば，図 11(a) は，図 10(b) で提示されている補完候補文の一覧のうち 2 番目の文が補完されたメソッドである．このとき，連続して補完することを考えると，図 10(b) で 1 番目に提示されていた文は補完された文よりも位置的に前に補完されるべきである．一方，図 10(b) で 3 番目以降に提示されていた文は補完された文よりも位置的に後ろに補完されるべきである．そのため，3 番目の以降の文は補完箇所を移動させる必要がある．実際，図 11(a) では補完された文とその直前の文に補完候補があることを示す下線が引かれている．また，補完された文の直前の文にある下線を選択すると，図 11(b) のように補完候補が提示される．図 11(b) で提示されている文は図 10(b) で 1 番目に提示されていた文である．

6 実験

本研究では，提案手法の有効性を評価するために被験者実験を行った．この章では，実験内容とその結果について述べる．

6.1 調査項目

提案手法の特徴は，未実装コードだけでなく書き漏らしコードも補完できることである．そのため，書き漏らしがどの程度の頻度で発生しているかが提案手法の有効性に影響を与える．また，未実装コードと書き漏らしコードの両方が実際に補完されるかどうかを検証する必要がある．

よって本研究では，被験者実験によって以下の 2 項目を調査する．

項目 1： 実装中に書き漏らしはどの程度発生するか.

項目 2： 提案手法は未実装コードと書き漏らしコードの両方を補完できるか.

6.2 実験方法

本研究で行った実験は大きく以下の 2STEP に分けられる．

STEP1： 被験者にいくつかのタスクを与え，被験者がそのタスクを実装する様子を録画する．

STEP2： 記録した映像を基に 6.1 で述べた 2 つの項目を調査する．

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．

STEP2 では，STEP1 で記録した映像を著者が見て 2 つの項目を調査する．項目 1 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．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．また，書き漏らしコードの発生状況を文単位で調査する．

項目 2 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．本研究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．また，補完を行う際にはあらかじめツールにソースコード集合を与えて補完用データベースを構築する必要がある．本研究では，*UCI source code data set* を用いる [24]．*UCI source code data set* は非常に大規模な Java のソースコード集合である．表 2 は *UCI source code data set* の構成を示している．

6.3 被験者

本実験の被験者は著者の所属する研究室の学生 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 人をのぞ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．

6.4 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．被験者は与えられた仕様以外に新しい機能を追加してもかまわない．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 3 は本実験で使用したタスクの概要を示している．

6.5 調査項目 1：書き漏らしはどの程度発生するか

項目 1 の調査結果を表 4 に示す．表 4 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ことを表

表 2: *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

表 3: 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	当てられた文字列配列を csv 形式で 1 つの文字列に連結する．

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 4 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 5 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 5 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．比較的に多い発生原因として「オブジェクトの生成，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 6.2 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．

表 4: 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

6.6 調査項目 2：提案手法は未実装コードと書き漏らしコードの両方を補完できるか

項目 2 の調査結果を表 6 に示す．表 6 は有益な補完候補が 1 文でも提示されたタスクの個数を示している．表の 2 列目は被験者が実際に実装した文をツールが提示したタスクの個数を表す．表の 3 列目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をツールが提示したタスクの個数を表す．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 6 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何らかの補完があったタスクの 31%に相当する．

表 7 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．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 7 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 7 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．

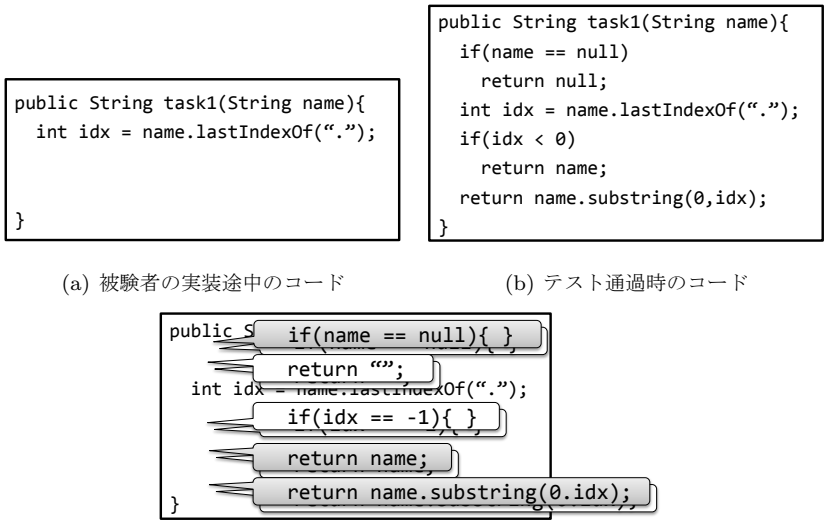
1 回の補完でいくつの文が補完されるかについても調査した．書き漏らしコードが補完される場合，

表 5: 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
オブジェクトの生成，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

表 6: 有益な補完候補が提示されたタスク

	被験者が実装した文を提示したタスク	被験者が実装した文と意味的に等しい文を提示したタスク	計	全タスク
全体	25	7	32	51
書き漏らし	8	2	10	41



(a) 被験者の実装途中のコード　　(b) テスト通過時のコード

(c) ツールによる補完候補文の提示

図 12: 有益な補完の例

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．

図 12 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 12(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 12(b) の実装に至った．一方で，図 12(a) のメソッドに補完を適

表 7: 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

用すると，図 12(c) のような補完候補文が提示される．図 12(b) と 12(c) を比較すると，図 12(c) の黒背景で表示されている補完候補文を被験者は実際に実装していることがわかる．

7　議論

7.1　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補完候補メソッドを基に補完候補文を提示している．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 [25, 26]．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集合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低下させることができると考えられる．

7.2　妥当性への脅威

本研究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

本研究の被験者は全員が著者と同じ研究室に所属している．そのため，別の大学の学生や産業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

本研究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研究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スーパー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集合に依存する．本研究の実験では，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集合の規模や構成を変えることによって結果が変わる可能性がある．

実装

本研究では，ツールの実装の際に正規化手法として *Parameterized Matiching*，順位付け手法としてクローンセットの要素数の考慮を行っている．一方で，正規化手法や順位付け手法は上記の手法以外にも様々な手法が存在する．そのため，正規化手法や順位付け手法を変えることによって結果が変化する可能性がある．

また，提案手法では文のハッシュ値を計算しハッシュ値を比較することによって文の同一性を確認している．しかし，ハッシュ値の衝突が起こっていれば，同一でない文を同一であると判定してしまう．そのため，ハッシュ値が衝突していた場合は結果に悪影響を与えている可能性がある．ただし，本研究で行った実験ではハッシュ値の衝突は確認されていない．

8 関連研究

文の補完

いくつかの既存研究では本研究と同様にコードの再利用を目的としたコード補完手法を提案している．Nguyen らは，複数のオープンソースソフトウェアから API の使用パターンを抽出し，その情報を基にコード補完を行うツール GraPacc を構築した [7]．GraPacc はあらかじめ複数のソフトウェアから API の使用パターンを取得する．取得した API の使用パターンはグラフベースでモデル化され，GraPacc はグラフから変数や制御構造などの特徴を抽出してデータベースに保存する．開発者は書きかけのコードを GraPacc に与えると，GraPacc は書きかけのコードから API の使用パターンを取得し，データベースにある最も類似度が高い API の使用パターンを補完する．

Hill らは記述途中のメソッドから特徴を抽出して，その特徴を基に自動的にメソッドを補完する手法を提案した [8]．この手法はフィンガープリントベースのコードクローン検出手法を用いて，補完候補となるメソッドを検索している．この手法では，メソッドの行数や引数また使われている識別子名をベクトル形式で表現する．あらかじめソースコード集合から上記のベクトルを計算しておき，補完対象となるメソッドから抽出したベクトルと最も類似したベクトルを持つメソッドを補完する．

上記の手法はいずれも文を補完するという観点で本研究と類似している．また，Hill らの手法は補完するメソッドの決定にコードクローン検出を用いている点も本研究と類似している．しかし，本研究とは異なり書き漏らしたコードの補完を目的としておらず，書き漏らしたコードが補完されることは稀である．また，いずれの手法も補完候補となるコードのパターンをそのまま補完する．一方で，本手法では複数の補完候補にある文を組み合わせて補完することができる．そのため，開発者が最適だと考えるコードの補完が実現できる．

Holmes らは実用的な再利用を支援する手法を提案している [9]．この手法は，開発者が再利用したいと考えている機能を見つけ出し，保存されている実用的な再利用情報を基に半自動的にその機能を補完する．Holmes らの手法はユーザが再利用したいと考えているコードをみつけて入力として与える必要があるのに対し，提案手法はあらかじめ多くのコードからコード片の情報を取り出してデータベースを構築するため，ユーザが知らないシステムからの再利用が可能である．

字句の補完

既存研究ではトークンの補完を効率的に行う手法も多く提案されている．Bruch らはデータセットの情報を学習して，その情報を基にメソッド呼び出しを補完する 3 つの手法 (*FreqCCS*, *ArCCS*, *BMNCCS*) を提案した [5]．*FreqCCS* は，データセット内で最も使用頻度が高いメソッド呼び出しを補完候補とする．*ArCCS* は，データセットからメソッド呼び出しの関係を抽出し，その関係を基に補完を行う．*BMNCCS* は，k 近傍法用いてデータセットを学習し，その学習データを用いて補完候補となるメソッド呼び出しを決定する．

Omar らは，補完候補を単純に選択する選択肢ベースの補完ではなくより高度なコード補完を支援

する枠組みを提案している [6]．ソフトウェア開発者へのインタビューを基に，Omar らは 2 つの高度な機能を持つコード補完システム *Graphite* を作成している．1 つ目の機能は色の補完で，グラフィカルに色の選択ができる他，色の名前で検索することも可能である．もう 1 つの機能は正規表現を記述支援で，開発者が正規表現を実際にコードとして記述する前にテストをすることが可能である．このテストでは，正規表現の構文エラーを見つけたり，テスト文字列を入力して実際に正規表現とマッチするかどうかを調べたりすることができる．その他にもコード補完手法がいくつか提案されている [10–12]．しかし，いずれもトークンの補完であり本研究の目的であるコード再利用支援とは注目点が異なる．

コード検索

本研究と同様にコードの再利用支援する手法としてコード検索が挙げられる．井上らは，関数の呼び出し関係からそれぞれの関数の重要度を計算するコンポーネントランク法とソースコード上の位置によってキーワードの重要度を変えるキーワードランク法を提案している [27, 28]．また，それらを実装したコード検索システムである *SPARS* を開発した [27]．コンポーネントランク法では，多くの関数から呼び出されている関数の重要度は高くなる．また，重要な関数から呼び出されている関数は重要度が高くなる．加えて，コンポーネントランク法では類似関数をグループ化して，要素の重要度の合計をグループの重要度として再定義している．キーワードランク法では，コードから抽出されたキーワードの重要度をそのトークンの種類から決定している．トークンの種類が重要であるほどキーワードの重要度が高くなる．例えば，メソッド名やクラス名から抽出されたキーワードは重要度が高くなる．

McMillan らは *Navigation Model* と *Association Model* という 2 つのモデルから関数の重要度を決定する手法を提案し，これらを実装した *Portfolio* を開発した [29]．*Navigation Model* は開発者がどのようにして関数をたどるかを表したモデルである．このモデルでは，関数の呼び出し関係を基にそれぞれの関数の重要度を計算しており，*PageRank* 手法を応用したものとなっている．*Association Model* はキーワード間の関連性を表したモデルである．このモデルでは，*Spreading activation* 法を用いてキーワード間の関連性を計算している．また，*Portfolio* はコードを提示する際に関数のコールグラフも提示する．ユーザはグラフをたどる事によってその関数の使い方を知ることができる．

その他にもコード検索手法は多くの研究者が提案している [30–33]．しかし，コード補完とは異なり開発者が必要な部分を探してそこをコピーする等の労力が必要になるため，実装効率の向上という観点ではコード補完に劣っている．

Type-3 コードクローン検出

本研究では，補完候補となるメソッドの特定に独自の Type-3 クローン検出手法を用いている．Type-3 クローン検出する既存手法として，Roy らはブロック単位の Type-3 クローンを検出するツール *NICAD* を開発している [34]．*NICAD* はブロック単位で最長共通部分列 (*LCS*) を計算し，閾値

以上の *LCS* を持つブロックの組をクローンとして検出する．しかし，*NICAD* は大量のソースコード集合からの検出に非常に時間が掛かる．また，本研究で使用する種類以外の Type-3 クローンも検出するため，使いたいクローンを特定するコストが必要になる．そのため，本研究では補完を目的とした高速な Type-3 クローン検出手法を独自に考案している．

9 おわりに

本研究では，開発者が書き漏らしたコードを補完することを目的した新しいコード補完手法を提案した．この手法では，補完の対象となるメソッドにある文をすべて含んだ Type-3 クローンを用いることで補完を実現している．この手法を使ってコード補完を行うことによって，開発者はこの後に実装するであろうコードだけでなく書き漏らしたコードを自動的に実装することができる．本研究では，提案手法の有効性を評価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，被験者がそのタスクを実装する流れを録画した．録画した映像を基に，書き漏らしコードの発生状況と提案手法の有効性を調査した．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．結果として，提案手法は途中まで実装されたコードの未実装部分と書き漏らしたであろうコードの両方で有益な補完候補を提示できていることがわかった．

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．本研究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はっきりすると考えられる．また，本研究で用いているランキング手法や正規化手法を変えることで結果にどのような変化が生まれるかを調査することも考えている．ランキング手法の例として，呼び出し回数の多いメソッドを上位にする手法を適用することが考えられる．また，正規化手法として，本研究では行っていない型名やメソッド呼び出しを正規化することが考えられる．

謝辞

本研究を行うにあたり，理解ある親身なご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，適切なご指導を賜り，有益かつ的確なご助言を多数頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究において，親切なご指導を頂き，多くのご助言，ご助力を頂きました 井垣 宏 特任准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を賜り，多大なるご助力を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

また，本研究に関して多くのご助言を頂くとともに，様々な面において親切なご助力，ご協力を頂きました楠本研究室の皆様に心より感謝致します.

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学大学院情報科学研究科，並びに大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] “Eclipse”. http://www.eclipse.org.

[2] “Intellisense”. http://blogs.msdn.com/b/vcblog/archive/tags/intellisense/.

[3] “Informer”. http://javascript.software.informer.com/downloadjavascript-code-completion-tool-for-eclipse-plugin/.

[4] 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.

[5] 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.

[6] 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.

[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. of the 34th International Conference on Software Engineering, pp.69–79, 2012.

[8] R. Hill and J. Rideout, “Automatic method completion,” Proc. of the 19th International Conference on Automated Software Engineering, pp.228–235, 2004.

[9] 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.

[10] R. Robbes and M. Lanza, “How program history can improve code completion,” Proc. of the 23rd International Conference on Automated Software Engineering, pp.317–326, 2008.

[11] C. Zhang, J. Yang, Y. Zhang, J. Fan, J. Zhao, and P. Ou, “Automatic parameter recommendation for practical api usage,” Proc. of the 34th International Conference on Software Engineering, pp.826–836, 2012.

[12] D. Hou and D.M. Pletcher, “Towards a better code completion system by api grouping, filtering, and popularity-based ranking,” Proc. of the 2nd International Workshop on Recommendation Systems for Software Engineering, pp.26–30, 2010.

[13] 井上克郎，神谷年洋，楠本真二，“コードクローン検出法,” コンピュータソフトウェア，vol.18, no.5，pp.47–54，2001.

[14] S. Bellon, "Detection of software clones," Technical Report, Institute for Software Technology, University of Stuttugart, 2003. available at http://www.bauhaus-stuttgart.de/clones/.

[15] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Transactions on Software Engineering, vol.31, no.10, pp.804–818, Oct. 2007.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol.28, no.7, pp.654–670, July 2002.

[17] I. Baxter, A. Yahin, M.A. L. Moura, and L. Bier, "Clone detection using abstract syntax trees," Proc. of the 14th International Conference on Software Maintenance, pp.368–377, Mar. 1998.

[18] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo, "Software analysis by code clones in open source software," Journal of Computer Information Systems, vol.XLV, no.3, pp.1–11, Apr. 2005.

[19] 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.

[20] "Java development tools". http://www.eclipse.org/jdt/.

[21] B.S. Baker, "On finding duplication and near-duplication in large software systems," Proc. of the 2nd Working Conference on Reverse Engineering, pp.86–95, 1995.

[22] R. Rivest, "The md5 message-digest algorithm," RFC 1321(Informational), Apr. 1992. http://www.ietf.org/rfc/rfc1321.txt.

[23] T. Ishihara, K. Hotta, Y. Higo, and S. Kusumoto, "Reusing reused code," Proc. of the 20th Working Conference on Reverse Engineering, pp.457–461, 2013.

[24] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "Uci source code data sets". http://www.ics.uci.edu/~lopes/datasets/.

[25] 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.

[26] 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.

[27] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," IEEE Transactions on Software Engineering, vol.31, no.3, pp.213–225, 2005.

[28] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: Relative significance rank for software component search," Proc. of the 25th International Conference on Software Engineering, pp.14–24, 2003.

[29] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usages," Proc. of the 33rd International Conference on Software Engineering, pp.111–120, 2011.

[30] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," Proc. of the 32nd International Conference on Software Engineering, pp.475–484, 2010.

[31] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," Proc. of the 22nd International Conference on Automated Software Engineering, pp.204–213, 2007.

[32] S.P. Reiss, "Semantics-based code search," Proc. of the 31st International Conference on Software Engineering, pp.243–253, 2009.

[33] N. Sahavechaphan and K. Claypool, "Xsnippet: mining for sample code," Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp.413–430, 2006.

[34] C.K. Roy and J.R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," Proc. of the 16th International Conference on Program Comprehension, pp.172–181, 2008.