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.

既存コードの再利用によるユーティリティメソッドの自動生成

松本淳之介† 　肥後　芳樹† 　下仲　健斗† 　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{j-matumt,higo,s-kento,kusumoto}@ist.osaka-u.ac.jp

あらまし　ユーティリティメソッドとは汎用的な処理をまとめたものであり，開発者がコーディングを行う際に頻繁に利用するものである．ユーティリティメソッドは実装を効率的に行うための補助的なメソッドであり，複数のプロジェクトにおいて共通して使用される．ユーティリティメソッドの自動生成が実現すれば，実装したい機能に必要なコーディング作業を軽減することができる．そこで本研究では，既存のユーティリティメソッドを再利用・加工することで，開発者が定めた仕様を満たすユーティリティメソッドを自動生成する手法を提案する．提案手法では，自動プログラム修正の技術を用いて既存のユーティリティメソッドを開発者が定めた仕様を満たすように加工する．さらに提案手法を用いてオープンソースソフトウェアに存在する既存のユーティリティメソッドを自動生成することができるかを実験し，16個のユーティリティメソッドの自動生成に成功した．

キーワード　ユーティリティメソッド，自動プログラム修正，自動生成

1. ま え が き

ユーティリティメソッドとは汎用的な処理をするメソッドのことであり，開発者がコーディングをする際，頻繁に利用するものである．例えば Java には java.util パッケージという汎用的な処理がまとめられたパッケージが用意されており，多くの開発者がこのパッケージを利用している．またユーティリティメソッドを集めた Guava [1] という Java プロジェクトは多くの開発者に利用されており，GitHub の利用者がお気に入りのオープンソースソフトウェアに送るスターの数が 20,000 を超えている．このことからもユーティリティメソッドが多くの開発者に利用されていることがわかる．

このように数多くのユーティリティメソッドが公開されているため，開発者は求めているユーティリティメソッドを探すのには労力が必要である．また，開発者が求めているユーティリティメソッドは必ず存在するわけではない．つまり，探して見つけることができなければ開発者自身の手で実装する必要がある．そのため開発者の立場からすれば，ユーティリティメソッドの探索およびその実装は開発をする上で手間のかかる作業といえる．

そこで本研究では開発者が実装したい機能に必要なコーディング作業を軽減できるようにユーティリティメソッドの自動生成を目的とする．ユーティリティメソッドは汎用的な処理をまとめたものであり，様々なプロジェクトで再利用される傾向にある．そこで本研究ではそのようなユーティリティメソッドの再利用性に注目する．自動生成したいメソッドと似た処理をする既存のユーティリティメソッドを再利用し，自動プログラム修正の技術で加工することで開発者が定めた仕様を満たすユーティリティメソッドを自動生成する手法を提案する．

似た手法を用いた既存の研究 [2] もあるが，その研究では実装したいメソッドと同一プロジェクト内のメソッドのみを再利用の対象として自動生成を行なう．つまり自動生成したいメソッドと似た処理をするメソッドが同一プロジェクト内に存在しなければ，自動生成をすることができない．本研究の提案手法では様々なプロジェクトに存在するメソッドを再利用して自動生成を行うため，より多くのメソッドを再利用の対象とし，自動生成をすることができる．

また，開発者が既存のメソッドを開発しているプロジェクトに適用する際の変更をパターン化し，プログラムに模倣させた研究も存在する [3]．この手法ではパターンに則った加工しかできず，自動プログラム修正の技術を用いた本研究のように柔軟にメソッドを加工することはできない．

2. 準　　備

本章ではまずユーティリティメソッドについて説明する．その後，遺伝的プログラミングを用いた GenProg という自動プログラム修正の手法について説明する．

2.1 ユーティリティメソッド

開発者は汎用的な処理をメソッドにまとめ，再利用することが頻繁にある．本研究では，そのような汎用的な処理をするメソッドをユーティリティメソッドとする．ユーティリティメソッドの例として，図 1 のようなものがある．これは leakcanary [4] という Java プロジェクトに含まれているメソッドであり，パッケージ名とクラス名で構成される完全限定名からクラス名だけを抽出するメソッドである．図 1 のソースコードからも分かる

```
String classSimpleName(String className) {
    int separator = className.lastIndexOf('.');
    if (separator == -1) {
        return className;
    } else {
        return className.substring(separator + 1);
    }
}
```

図 1　ユーティリティメソッドの例

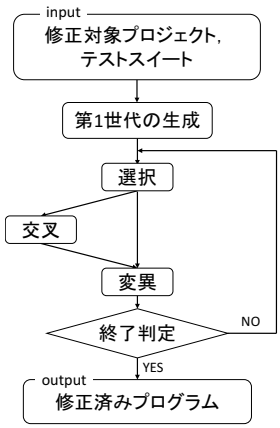

図 2　GenProg の動作の流れ

通り，このメソッドは特定のクラスに依存することなく，汎用的に使用することができる．

2.2 GenProg

GenProg は遺伝的プログラミング [5] に基づいて自動的にプログラムの修正を行う手法である [6]．GenProg は欠陥を含むプログラムおよびテストケースの集合であるテストスイートを入力として受け取り，再利用に基づく自動プログラム修正を行う．出力はテストスイートに含まれるすべてのテストケースを通過するプログラムである．ここで，入力として与える欠陥を含むプログラムのことを修正対象プログラム，出力として得られる修正が完了したプログラムのことを修正済みプログラムと呼ぶ．また，GenProg は自動プログラム修正を行う前に，入力されたテストケースを用いて修正対象プログラムの欠陥箇所の限局を行う．

GenProg の動作の流れを図 2 に示す．GenProg は欠陥箇所の限局を行った後，欠陥箇所に変異操作を行ったプログラム（以降，変異プログラムと呼ぶ）を複数生成する．これらの変異プログラム群のことを第 1 世代と呼ぶ．変異操作では，次の 3 処理のうちいずれか 1 つを行う．

挿入　欠陥を含む行の直後に，修正対象プログラムに含まれるプログラム文の挿入を行う操作

削除　欠陥を含む行を削除する操作

置換　修正対象プログラムからランダムに選択された行によって欠陥を含む行を上書きする操作（挿入操作＋削除操作）

次に，評価関数に基づいて各変異プログラムの評価値を計測し，評価値の高いものを一定数残し，それ以外を削除する．ここで残った変異プログラム群に対して交叉操作および変異操作を行うことで新たな変異プログラム群を得る．交叉操作は 2 つの変異プログラムを組み合わせることで新たな変異プログラムを生成する操作である．交叉操作および変異操作によって得られた変異プログラム群のことを次世代と呼ぶ．

次世代の変異プログラム群に対してテストを行い，すべてのテストケースを通過する変異プログラムがあれば，それを修正済みプログラムとして出力する．そのような変異プログラムが存在しなければ，選択操作からやり直す．この処理をすべてのテストケースを通過する変異プログラムが生成されるか，あらかじめ定められた世代数に到達するまで繰り返す．GenProg では，ソースコード中に存在する行を用いて欠陥を修正できると仮定している．この仮定の正しさを検証するために，Barr らは実際に行われた変更を基に調査を行った [7]．調査の結果，ソースコード中の行を用いることで，10%の変更において追加された行のすべての行を記述することができ，42%の変更において追加された半数以上の行を記述できることが分かった．

3. 提 案 手 法

本研究では，既存の Java メソッドを再利用し，ユーティリティメソッドを自動生成する手法を提案する．提案手法の概要を図 3 に示す．本研究における提案手法の入力は，既存の Java プロジェクト群，生成したいメソッドのメソッド名と返り値と引数の型 (以降，返り値と引数の型の組み合わせをシグネチャと呼ぶ)，そして生成したいメソッドに関するテストケースである．出力は仕様を満たすメソッドである．本研究の提案手法は以下の 2 つの工程で構成される．

- 前準備
- メソッドの自動生成

前準備として既存の Java プロジェクト群からメソッドの情報を抽出し，メソッドの情報を格納するデータベース (以降データベースのことを DB と呼ぶ) を構成する．この前準備はメソッドの自動生成を行う度に実行する必要はない．

メソッドの生成は次の 3 ステップから構成される．

Step 1: メソッドの検索
Step 2: メソッド群の並び替え
Step 3: メソッドの加工

Step 1 では，生成したい Java メソッドと同一のシグネチャのメソッドを DB から取り出す．Step 2 では，Step 1 で取り出したメソッド群の並び替えを行う．並び替えを行うのは，DB に膨大な数のメソッドが登録されている場合，無作為に加工するメソッドを選択すると生成に時間がかかってしまうからである．Step 3 では，Step 2 で並び替えたメソッドを先頭から順番にテストケースを通過するよう加工していく．

4. 実　　装

4.1 前 準 備

前準備として既存の Java プロジェクトから Java メソッドを抽出し，DB に登録していく．DB として SQLite [8] を用いた．

DB にメソッドを登録する際，Java プロジェクトに属している全ての Java メソッドを DB に登録するわけではない．本研究で生成したいメソッドはユーティリティメソッドであるため，DB に登録するメソッドもユーティリティメソッドに限定した．異なるプロジェクト間でユーティリティメソッドをコピーし，加工するためにも，本研究で対象にするユーティリティメソッドは

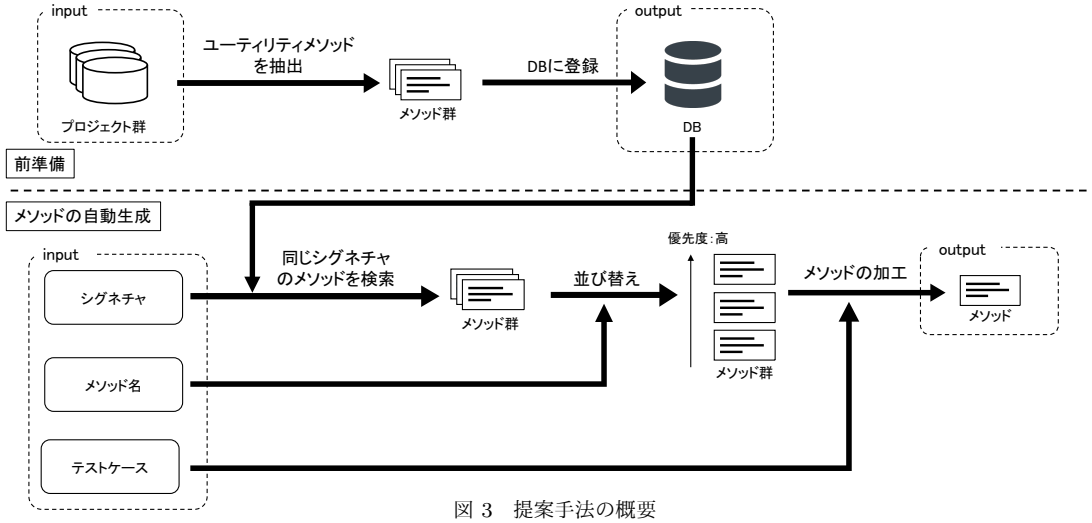

図 3　提案手法の概要

次の特徴をもつメソッドである．

- プリミティブ型，もしくは java.lang. パッケージのクラスにしかアクセスしていない
- 引数もしくはローカルな変数にしかアクセスしていない
- java.lang パッケージのクラスに定義されているメソッド，もしくは本研究で対象にするユーティリティメソッドしかメソッドを呼び出していない

また，DB に登録するメソッドの情報は以下のものである．

- メソッドの名前
- 引数の型
- 返り値の型
- ソースコード
- 内部で呼び出しているユーティリティメソッド

4.2　メソッドの自動生成

Step 1: メソッドの検索

Step 1 では自動生成したいメソッドと同一のシグネチャを持つメソッド群を DB から取り出す．

Step 2: メソッド群の並び替え

Step 1 で取り出したメソッド群を一つずつ仕様を満たすよう Step 3 で加工する際，全てのメソッドが仕様を満たすよう加工できるわけではない．本研究では，自動生成したいメソッドの名前と加工対象のメソッドの名前との類似度が高いほど似た処理をするという先行研究 [9] [10] を踏まえ，メソッドの名前の類似度が高い順番にメソッドの並び替えを行う．加工に成功する可能性の高いメソッドを早い段階で加工の対象にすることで，実行効率をあげることができる．メソッド名の類似度は，レーベンシュタイン距離を用いて計算した．

Step 3: メソッドの加工

Step 2 で並び替えたメソッド群を先頭から順番に加工していく．加工の手順は次の通りである．

1. 加工対象のメソッドのソースコードを定義したいクラスにコピーする．この時，DB から取り出したメソッドの処理の内部で別のユーティリティメソッドを呼び出している場合は自動生成したいメソッドが定義されるクラスに追加しておく．
2. GenProg を用いてメソッドがテストケースを通過するまで加工していく．

仕様を満たすようメソッドを加工することができれば加工されたメソッドを出力する．加工することができなければ，Step 2 で並び替えたメソッド群から次のメソッドを取り出し，上の手順で再度加工させていく．

メソッドをコピーして加工する例を図 4 に示す．この例は，

- 2 つの int を引数に持ち int を返り値とするシグネチャ
- “min” という文字列

の 2 つを入力としており，引数で与えられた数値の大きい方を返す max メソッドを用いて，引数で与えられた数値の小さい方を返す min メソッドを生成する例である．この例ではまず max メソッドのソースコードを min メソッドにコピーし，それを GenProg を用いて小さい値を返す処理になるよう加工している．

5.　適 用 実 験

本章では，提案手法を評価するために行った実験と，その実験結果について述べる．

5.1　実 験 概 要

既存の Java プロジェクト内に含まれるユーティリティメソッドを，提案手法を用いて自動生成できるか確かめた．

5.2　実 験 対 象

5.2.1　メソッドを格納する DB の構築

GitHub に公開されているスター上位の Java プロジェクト 30 個に加えて，“apache/commons”, “math”, “util”, “algorithm”, “competitive” という単語で検索して得られた計 209 の Java プロジェクトを対象にした．これらの Java プロジェクトを入力とし，その結果 1,821 個のメソッドが DB に格納された．

5.2.2　自動生成対象のメソッド

実験をするにあたり，どのプロジェクトのどのメソッドを自動生成の対象とするか決める必要がある．実験を自動化するにあたり，先に述べた 209 のプロジェクトから以下の 2 つの条件を同時に満たすプロジェクトを対象にした．

- Maven [11] で管理されている
- JaCoCo [12] でカバレッジレポートを自動生成できる

上記の条件を満たすプロジェクトに定義されているメソッドのうち，以下の特徴をもつメソッドを自動生成対象とした．

- 本研究で対象にしているユーティリティメソッドである

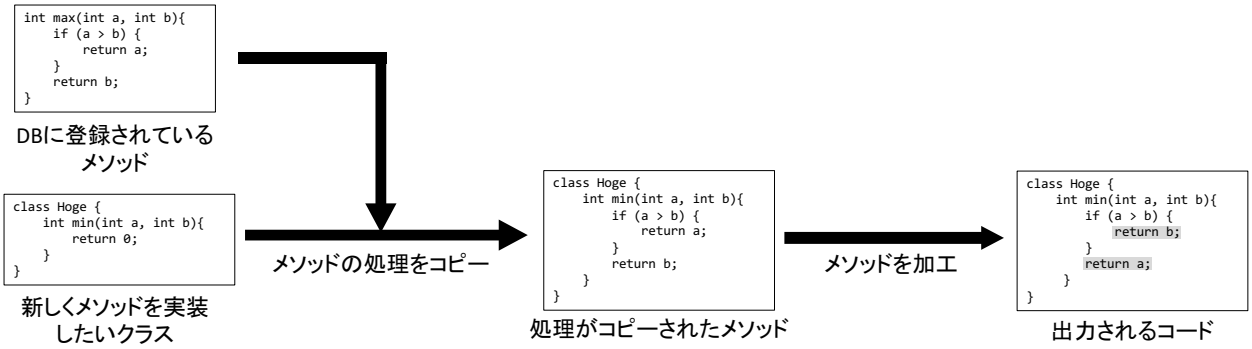

図 4　メソッドをコピーして加工する流れ

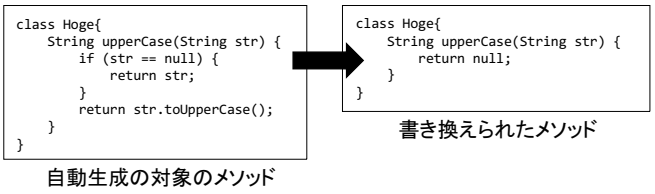

図 5　対象のメソッドの書き換え例

- テストカバレッジの値が 1 である

ユーティリティメソッドの生成が可能であるかを確かめる実験であるため，自動生成対象のメソッドは本研究で対象にしているユーティリティメソッドに限定した．また，本研究の提案手法では自動生成をするにあたり生成したいメソッドのテストケースが必要であることから，テストカバレッジの値が高いものに限定した．テストカバレッジの自動生成を行うために，実験対象のプロジェクトは JaCoCo でカバレッジレポートを自動生成できるものに限定した．このような特徴をもつメソッドを 176 個見つけることができた．

5.3　実 験 手 順

自動生成対象のメソッドに対して以下の手順で実験を行なった．

1. 対象のメソッドが定義されている Java プロジェクトのテストを実行する．
2. 対象のメソッドの内容をでたらめな処理に置換する．
3. 再度テストを実行する．
4. 1. でテストを実行した際には成功したにも関わらず 3. でテストを実行した際には失敗したテストを特定する．
5. 失敗したテストを入力とし，提案手法を用いてメソッドの自動生成を試みる．
6. 出力されたメソッドが正しく自動生成に成功しているか目視で確認する．

2. のメソッドの書き換えについては，そのメソッドの返り値の型に応じた書き換えをする．例えば図 5 のように返り値が参照型であればメソッドの処理が"return null;"のみになるよう書き換える．返り値の型が int などのプリミティブな型であれば，その型に対応する任意の定数 (例えば int なら 0) を返す処理に書き換える．このような処理の書き換えをすることで，コンパイルには成功するがテストに失敗することが予想される．もし 3. のテストに成功してしまった場合，提案手法の入力となるテストケースを特定できないので，そのメソッドでは実験を行わない．

実験をするにあたり，自動生成対象のメソッドと同一のメソッドがすでに DB に登録されている可能性があるため，自動生成の対象のメソッドと完全限定名が等しいメソッドは加工対象として取り除いてある．また効率的よく実験を行うために，1 つの自動生成の対象のメソッドに対して加工対象のメソッドは 10 個までという制限を設けた．つまり提案手法の Step 2 で並び替えたメソッド群の先頭から 10 番目までしか提案手法の Step 3 の加工対象にしかならず，10 番目のメソッド加工に失敗した場合は自動生成に失敗したものとして扱う．

5.4　出力されるメソッドの分類

出力されるメソッドは次のように分類する．

再利用されただけのメソッド　再利用するメソッドを加工せずにテストケースを通過し，開発者が意図した処理をするメソッド

加工されたメソッド　再利用するメソッドは加工しない状態ではテストケースを通過しないが，加工することでテストケースを通過し，開発者の意図した処理をするメソッド

オーバーフィットなメソッド　テストケースを通過することはできるが，開発者の意図から外れた処理をするメソッド

この実験では，自動生成に成功したメソッドは再利用されただけのメソッド，及び加工されたメソッドである．オーバーフィットなメソッドは自動生成に失敗したとして扱う．

ただし，自動生成対象のメソッドと出力されたメソッドの差異が，引数の null チェックの有無のみである場合は，出力されたメソッドは開発者の意図した処理をするものとして扱った．

5.5　実 験 結 果

実験の結果として 18 個のメソッドが出力された．出力されたメソッドの内訳を表 1 に示す．以降では再利用されただけのメソッド，加工されたメソッド，オーバーフィットなメソッドが出力されたそれぞれの例を紹介する．

再利用されただけのメソッドの例

自動生成対象のメソッドは Algorithms [13] というプロジェクトに存在している calculateSum メソッドである．このメソッドのソースコードと出力されたメソッドのソースコードを図 6 に示す．このメソッドは引数で与えられた int 型の配列の総和を計算するメソッドである．このメソッドを生成するにあたり，zxing [14] というプロジェクトに存在している sum メソッドを再利用した．この例では加工する必要がなかったため，再利用されたメソッドと出力されたメソッドは完全に同一である．よっ

表 1　出力されたメソッドの内訳

再利用されただけのメソッド	14
加工されたメソッド	2
オーバーフィットなメソッド	2

```
int calculateSum(int[] numbers) {
    int sum=0;
    for (int n : numbers) {
        sum+=n;
    }
    return sum;
}
```

(a) 自動生成の対象となるメソッド

```
int calculateSum(int[] array) {
    int count=0;
    for (int a : array) {
        count+=a;
    }
    return count;
}
```

(b) 出力されたメソッド

図 6　再利用されただけのメソッドの例

```
String uncapitalize(String str) {
    int strLen;
    if (str == null || (strLen=str.length()) == 0) {
          return str;
    }
    final int firstCodepoint = str.codePointAt(0);
    final int newCodePoint = Character.toLowerCase(firstCodepoint);
    if (firstCodepoint == newCodePoint) {
        return str;
    }
    final int newCodePoints[]= new int[strLen];
    int outOffset=0;
    newCodePoints[outOffset++]=newCodePoint;
    for (int inOffset=Character.charCount(firstCodepoint); inOffset < strLen; ) {
        final int codepoint = str.codePointAt(inOffset);
        newCodePoints[outOffset++]=codepoint;
        inOffset+=Character.charCount(codepoint);
    }
    return new String(newCodePoints,0,outOffset);
}
```

(a) 自動生成の対象となるメソッド

```
String uncapitalize(String name) {
    if (name == null || name.length() == 0) {
        return name;
    }
    char chars[]=name.toCharArray();
    chars[0]=Character.toLowerCase(chars[0]);
    return new String(chars);
}
```

(b) 出力されたメソッド

```
String decapitalize(String name) {
    if (name == null || name.length() == 0) {
        return name;
    }
    if (name.length() > 1 && Character.isUpperCase(name.charAt(1))
                          && Character.isUpperCase(name.charAt(0))) {
        return name;
    }
    char chars[]=name.toCharArray();
    chars[0]=Character.toLowerCase(chars[0]);
    return new String(chars);
}
```

(c) 加工する際に再利用されたメソッド

図 7　加工されたメソッドの例

て，この例は再利用されただけのメソッドに分類する．

加工されたメソッドの例

自動生成対象のメソッドは commons-lang [15] というプロジェクトに存在している uncapitalize メソッドである．このメソッドは引数で与えられた文字列の先頭の文字を小文字にした文字列を返すメソッドである．このメソッドのソースコード，出力されたメソッドのソースコードと加工する際に再利用されたメソッドのソースコードを図 7 に示す．このメソッドを生成するにあたり，fastjson [16] というプロジェクトに存在している decapitalize メソッドを加工した．この例ではメソッドを再利用するだけではテストケースを通過することができなかったため，GenProg によって加工が施された．図 7 の (b) と比べると，図 7 の (c) で示された網かけの部分のコードが削除されて出力されていることがわかる．再利用されたメソッドに加工が加えられているので，この例は加工されたメソッドに分類する．

オーバーフィットなメソッドの例

自動生成の対象となるメソッドは commons-jxpath [17] というプロジェクトに存在している equalStrings メソッドである．

```
boolean equalStrings(String s1, String s2) {
    if (s1 == s2) {
        return true;
    }
    s1=s1 == null ? "" : s1.trim();
    s2=s2 == null ? "" : s2.trim();
    return s1.equals(s2);
}
```

(a) 自動生成の対象となるメソッド

```
boolean equalStrings(String s1, String s2) {
    if (s1 == null && s2 == null) return true;
    if (s1 == null || s2 == null) return false;
    return s1.equals(s2);
}
```

(b) 出力されたメソッド

図 8　オーバーフィットなメソッドの例

このメソッドは引数で与えられた 2 つの文字列に対して，trim メソッドを呼び出し，その結果が等しいかどうか判断するメソッドである．このメソッドのソースコード，出力されたメソッドのソースコードを図 8 に示す．このメソッドを生成するにあたり，dubbo [18] というプロジェクトの isEquals メソッドが再利用した．この例では加工することなくてテストケースを通過したため，再利用されたメソッドと出力されたメソッドは完全に同一である．しかし，図 8 の (a) と (b) のソースコードを比べるとわかるように，自動生成の対象のメソッドと出力されたメソッドの処理は trim メソッドの呼び出しの有無が異なっており，出力されたメソッドはオーバーフィットなメソッドであることがわかる．

6. 考　　察

本章では，5 章で述べた適用実験についての考察を行う．

6.1 再利用されただけのメソッドに対する考察

再利用されただけのメソッドが多数出力された理由として，本研究で対象にしているユーティリティメソッドが限定的なことが考えられる．本研究では対象にするメソッドを 4.1 章で述べた特徴を持つものに限定している．4.1 章で述べた特徴を持つメソッドの処理の種類が少なく，同一の処理をするメソッドが多数収集された．再利用されたメソッドが出力されるのは収集したメソッド内に同一の処理をするメソッドが複数あった場合であるので，対象にしているメソッドの種類が少なくなったことで再利用されただけのメソッドが多数出力されたと考えられる．

開発者の立場で考えると仕様を入力するだけでメソッドが生成されることが重要である．そのメソッドが既存のメソッドから加工されたかどうかは重要ではない．再利用されただけのメソッドが出力されることは本研究の実験としては成功である．

6.2 加工されたメソッドに対する考察

再利用されたメソッドが多く出力されたことに対して，加工されたメソッドは非常に少ない結果となった．これに対する原因として，次の二点が考えられる．

- 自動生成対象のメソッドのソースコードと似たソースコードのメソッドが加工対象のメソッドになかった
- GenProg の精度が十分でなかった

本研究の提案手法では，自動生成の対象のメソッドと似た処理をするメソッドに対して，GenProg を用いて加工を行う．そのため，自動生成の対象のメソッドと似た処理を行うメソッドを収集

し，加工対象にする必要がある．このようなメソッドを加工対象に含められなかったことが原因の一つと考えられる．

また，GenProg の精度の問題がある．GenProg のバグ修正の技術は既存の 105 個のバグに対して適用され 55 個の加工に留まっていることがわかっている [6]．今後の GenProg の発展に応じて，加工されたメソッドの出力数も増えることが考えられる．

ただし，加工されたメソッドの数が少ないこと自体は問題ではない．加工されたメソッドを出力する場合，加工するための時間が膨大となることがある．また，加工されたメソッドは必ずしも人間に読みやすいコードとなっているわけではない．開発者の立場で考えると，実行時間や可読性の観点で，加工されたメソッドが出力されるよりもそのまま再利用されたメソッドが出力された方が優れた出力と言える．

6.3 オーバーフィットなメソッドに対する考察

自動生成に失敗した原因としてテストケースの不足が考えられる．今回の実験をするにあたり，実験対象のメソッドを少しでも多く増やして実験するため，アクセス修飾子が private なものも含めている．private なメソッドは直接的にはテストの対象にならないため，そのメソッドは間接的に他の public なメソッドの内部で呼び出されることでテストされていることになる．つまり，private なメソッドに対して開発者が直接仕様を決めてテストケースを用意したわけではないため，十分なテストケースが用意されずにオーバーフィットなメソッドを生成してしまったと考えられる．

6.4 全体の出力に対する考察

加工されたメソッドの例で示したように，自動生成対象のメソッドのソースコードと比べて出力されたメソッドのソースコードは非常にコンパクトなものとなった．本研究の提案手法の場合，入力されたテストケースの入出力の組み合わせが一致していればメソッドの生成に成功したものとして扱うため，メソッドの処理の内部でのアルゴリズムまで指定して生成することができない．その結果，今回のように結果として自動生成の対象のメソッドよりも短いソースコードのメソッドの生成に成功する場合も考えられる．今回の実験では生成されなかったが，逆に自動生成の対象のメソッドよりも長いソースコードのメソッドが出力される場合も考えられる．

7. あとがき

本研究では，既存のユーティリティメソッドを再利用・加工することでユーティリティメソッドの生成を行い，提案手法でメソッドが生成できるか実験を行った．再利用されるメソッドが多く，加工されたメソッドが少ない結果になったが，合計して 16 のメソッドの生成に成功した．

今後の課題としては次のようなものが考えられる．

再利用対象のメソッドの拡張： 本研究では，ユーティリティメソッドにいくつかの条件を設けた．そのため生成が可能なメソッドが比較的単純なものになってしまった．より複雑な処理を持ったメソッドを再利用の対象にすることでより複雑な処理を持ったメソッドの生成が可能になる．

メソッドの並び替え： 本研究では，加工対象のメソッドが複数存在した場合は，メソッド名の類似度で並び替えを行った．しかし，このメソッド名の類似度による並び替えの場合，テストケースを通過することができるメソッドであるにも関わらずメソッド名の類似度が低いと，後ろの方に並び替えられてしまう．その場合，実行時間が長くなってしまう上，今回行った実験では先頭から 10 番目に入らなければ実験に失敗したと扱われてしまう．より効率的なメソッドの並び替えを行うことで実験の成功率も上がると予想される．

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：JP25220003）の助成を得て行われた．

文　献

[1] “Github - google core libraries for java”. https://github.com/google/guava
[2] 下仲健斗，肥後芳樹，松本淳之介，内藤圭吾，楠本真二，“シグネチャ情報と入出力情報を用いた java メソッドの生成，” 電子情報通信学会技術研究報告，vol.117，no.380，pp.007–012，Jan. 2018.
[3] S.P. Reiss, “Semantics-based code search,” Proceedings of the 31st International Conference on Software Engineering, pp.243–253, ICSE ’09, IEEE Computer Society, Washington, DC, USA, 2009. http://dx.doi.org/10.1109/ICSE.2009.5070525
[4] “Github - a memory leak detection library for android and java.”. https://github.com/square/leakcanary
[5] J.R. Koza, Genetic programming: on the programming of computers by means of natural selection, vol.1, MIT press, 1992.
[6] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” Proceedings of the 34th International Conference on Software Engineering, pp.3–13, ICSE ’12, IEEE Press, Piscataway, NJ, USA, 2012. http://dl.acm.org/citation.cfm?id=2337223.2337225
[7] E.T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, “The plastic surgery hypothesis,” Proceedings of the International Symposium on Foundations of Software Engineering, pp.306–317, 2014.
[8] “SQLite”. https://www.sqlite.org
[9] A. Corazza, S.D. Martino, V. Maggio, and G. Scanniello, “Investigating the use of lexical information for software system clustering,” Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, pp.35–44, CSMR ’11, IEEE Computer Society, Washington, DC, USA, 2011. http://dx.doi.org/10.1109/CSMR.2011.8
[10] Y. Higo and S. Kusumoto, “How should we measure functional sameness from program source code? an exploratory study on java methods,” Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.294–305, FSE 2014, ACM, New York, NY, USA, 2014. http://doi.acm.org/10.1145/2635868.2635886
[11] “Maven - welcome to apache maven”. https://maven.apache.org
[12] “JaCoCo - Java Code Coverage Library”. http://www.eclemma.org/jacoco/trunk/index.html
[13] “GitHub - Solutions for some common algorithm problems written in Java.”. https://github.com/pedrovgs/Algorithms
[14] “GitHub - ZXing (Zebra Crossing) barcode scanning library for Java, Android”. https://github.com/zxing/zxing
[15] “GitHub - Mirror of Apache Commons Lang”. https://github.com/apache/commons-lang
[16] “GitHub - A fast JSON parser/generator for Java”. https://github.com/alibaba/fastjson
[17] “Github - mirror of apache commons jxpath”. https://github.com/apache/commons-jxpath
[18] “Github - dubbo is a high-performance, java based, open source rpc framework”. http://dubbo.io