既存メソッドの再利用・加工による自動プログラミング

松本 淳之介[1,a)]　肥後 芳樹[1,b)]　下仲 健斗[1,c)]　楠本 真二[1,d)]

概要：ソースコードを自動的に生成する，自動プログラミングと呼ばれる技術は古くから研究されている．自動プログラミングとは，断片的な情報からプログラムを自動で生成する技術である．自動プログラミングにおいて課題となるのは，開発者の意図を断片的な情報としてどのように表現するか，である．しかし既存手法には，用途が限定的である，開発者の意図の表現があいまいすぎる，などの課題が多く残っている．本論文では，自動生成の対象を Java メソッドに限定し，Java メソッドの仕様からソースコードを自動生成することを試みる．仕様は，シグネチャ情報（引数の型と返り値の型，メソッド名）および入出力情報（引数の値と返り値の組の集合）である．提案手法では，シグネチャ情報を用いて既存の Java ソースコードを探索し，生成する Java メソッドの基となりうるコードを発見する．そして，入出力情報を満たすようにコードを加工する．提案手法を評価するために，オープンソースに存在するメソッドに対して本手法を適用し，37 個のメソッドを自動生成することに成功した．

キーワード：自動プログラミング，自動プログラム修正，メソッドの再利用

1. まえがき

自動プログラミングとは，プログラムを自動的に生成する技術のことである．つまり，開発者が直接プログラムを書くのではなく，作成したいプログラムの挙動や仕様を断片的に与えると，それを満たすプログラムを出力する技術である．

入力としてプログラムの概形を与え，細部を自動的に埋めてくれるスケッチングという手法が提案されている [1]．プログラムの概形とは，コードの一部 (定数など) が欠けている状態のプログラムである．開発者は，コードを完全に書く必要がなく，書かれていない部分が自動的に補われる．スケッチングを用いる場合，開発者はプログラムの概形と同時にテストケースも与える必要がある．スケッチングは，そのテストケースを満たすように SMT ソルバを用いて書かれていない部分を補う．

また，自然言語を入力とする自動プログラミング手法も提案されている．Gvero らは，関数呼び出しの自動補完ツールである AnyCode を開発した [2]．AnyCode では，関数名を入力するのではなく，呼び出したい関数の処理内容を自然言語で入力する．入力された自然言語を AnyCode が解析し，開発者の意図に適した関数の候補が提示される．

Galuwani らは，文字列操作の自動化ツールである FlashFill[*1] を開発した [3]．FlashFill は Microsoft 社製のスプレッドシートソフトウェアである Excel に実装されている機能であり，開発者が与えた入出力例から，開発者の意図を推量し，自動で文字列操作を行う．

自動プログラミングの手法はこれまでいくつか提案されてきているが，次の 3 つの理由から実用化が困難といわれている [4]．

- 開発者の意図を入力として表現する難しさ
- 候補となるプログラムを抽出する難しさ
- 大規模なプログラム作成の難しさ

本論文では，Java メソッドを自動生成の対象とし，Java メソッドの仕様からソースコードを自動生成することを提案する．入力として与える Java メソッドの仕様は以下の 2 種類とする．

- シグネチャ情報 (引数の型，返り値の型，メソッド名)
- 入出力情報 (引数の値と返り値の組の集合)

Java メソッドにおいて，一般的にシグネチャとはメソッド名と引数の型を指すが，本論文では返り値の型も含めてシグネチャとする．ソフトウェア開発のプロセスでは，ソ

[1] 大阪大学大学院情報科学研究科
[a)] j-matumt@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp
[c)] s-kento@ist.osaka-u.ac.jp
[d)] kusumoto@ist.osaka-u.ac.jp

[*1] https://goo.gl/PNLmfc

フトウェアの要件定義および設計に始まり，コーディングとテストを行い，運用と保守のフェーズに入る．このプロセスはウォーターフォールと呼ばれ，ソフトウェア開発における最も有名な開発モデルである．本手法の入力は，ソフトウェア開発のプロセスにより定まる仕様であり，提案手法はコーディングフェーズを自動化する手法といえる．提案手法の流れとしては，シグネチャ情報を用いて既存の Java ソースコードを探索し，生成する Java メソッドの基となりうるコードを発見する．そして，入出力情報 (以降テストケースと呼ぶ) を満たすようにコードを加工する．提案手法が入力として必要とするシグネチャ情報はソフトウェア開発の設計フェーズで作成される情報であり，入出力情報は単体テストのフェーズで作成される情報である．つまり，提案手法を用いるために開発者は新しく情報を用意する必要はない．

この提案手法を評価するために，既存のオープンソースのメソッドを自動生成することができるか実験を行い，37 個のメソッドを自動生成することに成功した．

2. 準備

本論文ではメソッドの自動生成を行うために，自動プログラム修正ツールを利用する．本章では自動プログラム修正ツールで用いられる欠陥箇所の限局について説明した後，自動プログラム修正ツールである GenProg について説明する．

2.1 欠陥箇所の限局

デバッグを支援する手法の 1 つに欠陥箇所の限局がある．欠陥箇所の限局手法はソースコードを解析して欠陥箇所の候補を探す手法であり，欠陥の可能性が高い箇所を開発者に提示することでデバッグを補助する．欠陥箇所の限局手法には，テストケース毎の実行パスから算出した疑惑値に基づいて順序付けを行う手法が存在する [5]．疑惑値とは，欠陥が存在する可能性の高さを表す値である．このような手法では，成功したテストのみが実行する文は疑惑値が低くなり，失敗したテストのみが実行する文は疑惑値が高くなる．また，後述する GenProg のようなプログラムの変更を行う手法のいくつかは，欠陥箇所の限局を行い，疑惑値の高いプログラム文を優先的に変更している．

2.2 GenProg

GenProg は遺伝的プログラミング [6] 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 [7]．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．

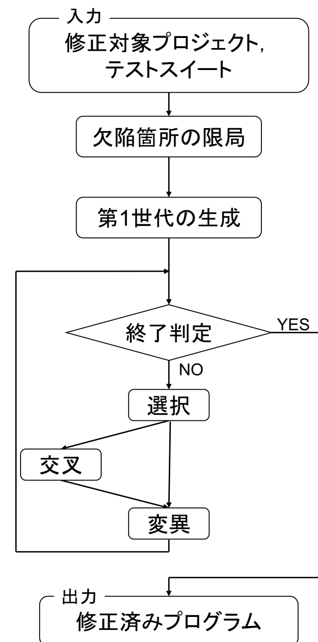

図 1: GenProg の動作の流れ

GenProg の動作の流れを図 1 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 欠陥を含んでいる可能性のある行の直後に，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処理

削除 欠陥を含んでいる可能性のある行を削除する処理

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んでいる可能性のある行を上書きする処理（挿入処理＋削除処理）

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．

GenProg では成功したテストの数を評価値として用いている．ここで残った変異プログラム群に対して一部のプログラム群だけ交叉操作をした後，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

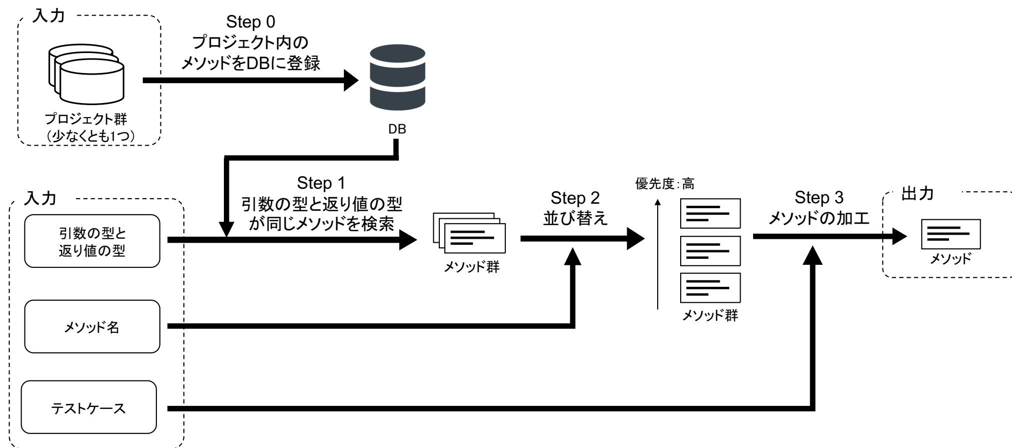

図 2: 提案手法の概要

プログラムが存在しなければ，次世代の変異プログラムを生成する．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が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 [8]．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

本論文では，既存のメソッドを再利用・加工することにより，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 2 に示す．本論文における提案手法の入力は以下の 3 つである．

- 既存のプロジェクト群 (少なくとも 1 つ)
- 自動生成したいメソッドのシグネチャ
- 自動生成したいメソッドのテストケース

出力は仕様を満たすメソッドである．

本論文の提案手法は以下のステップから構成される．

Step 0: 前準備
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 0 では，前準備として既存のプロジェクト群からメソッドの情報を抽出し，データベース (以降データベースのことを DB と呼ぶ) に登録する．この前準備は最初に一度だけ実行すれば良い．

Step 1 では，前準備で得られた DB から自動生成したい Java メソッドと返り値の型と引数の型が同じメソッドを取り出す．

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．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．加工に成功する可能性の高いメソッドを高い優先度で加工の対象にすることで，より短い時間でメソッドの自動生成ができると著者らは考えたため，メソッドの加工をする前にメソッド群の並び替えを行なっている．

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

Step 0: 前準備

前準備として既存のプロジェクトからメソッドを抽出し，DB に登録する．DB として SQLite*2を用いた．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

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返り値の型と引数の型が同じメソッド群を DB から取り出す．

Step 2: メソッド群の並び替え

本論文では，メソッドの名前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，自動生成したいメソッドと名前の類似度が高い順番にメソッド群の並び替えを行う．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する．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自動生成したいクラスにコピーする．
(2) GenProg を用いてメソッドが全てのテストケースを通過するまで加工する．

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

*2 https://www.sqlite.org

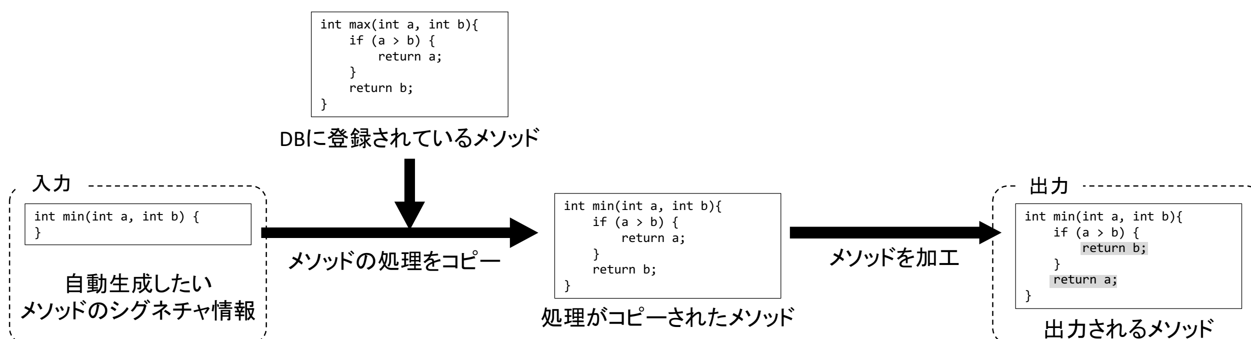

図 3: メソッドをコピーして加工する流れ

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する．

また，本論文が Java を対象としているの GenProg が Java を対象としているためである．

メソッドをコピーして加工する例を図 3 に示す．これは以下のシグネチャを入力とし，引数で与えられた数値の小さい方を返す min メソッドを自動生成する例である．

メソッド名:　“min”
引数の型：　2 つの int
返り値の型：　int

この例ではまず，引数で与えられた数値の大きい方を返す 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

本章では既存のオープンソースプロジェクトに対して行った適用実験の内容とその結果について述べる．本実験の目的は 3 章で述べたメソッドの自動生成が既存のプロジェクトでどの程度成功するのか調査することである．

以下の 2 つの実験を行なった．

[実験 A]　様々なプロジェクトのユーティリティメソッドを基にメソッドを自動生成する実験

[実験 B]　同一プロジェクトのメソッドを基にメソッドを自動生成する実験

2 つの実験を行ったのは，この 2 つのコンテキストで提案手法が利用されることを著者らが想定しているからである．

また，4 章で述べたように本論文では自動プログラム修正ツールとして GenProg を用いている．実験 B では，GenProg 以外の自動プログラム修正ツールと比較するため GenProg だけでなく Nopol[11] という自動プログラム修正ツールを用いても実験を行なった．

以降，GenProg を用いて行った実験 B のことを実験 B-1，Nopol を用いて行った実験 B のことを実験 B-2 とする．

5.1 実験 A について

5.1.1 DB の構築

様々なプロジェクトのユーティリティメソッドを用いて DB を構築する．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 個のプロジェクトに定義されたメソッドのうち，さらに次のような特徴をもつメソッドに限定した．

- 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，もしくは本実験で対象にするユーティリティメソッドしか呼び出していない．

このような特徴を持つメソッドが上で集めたプロジェクト群に 1,821 個含まれており，その全てを DB に登録した．本実験ではこれらの特徴を持つメソッドをユーティリティメソッドとして扱う．

5.1.2 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 個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[*3] で管理されている
- JaCoCo[*4]でカバレッジレポートを自動生成できる

上記の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．

- ユーティリティメソッドである．
- 対象プロジェクトに付随するテストケースの命令カバレッジとブランチカバレッジがともに 100% である

本論文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

[*3] https://maven.apache.org
[*4] http://www.eclemma.org/jacoco/trunk/index.html

自動生成を行うために，JaCoCo でカバレッジレポートを自動生成できるプロジェクトに限定した．

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．この 176 個のメソッドを提案手法で自動生成できるかを確認した．DB に登録されたメソッドが自動生成対象のメソッドになる場合もあるので，実験する時は完全限定名が一致するメソッドを再利用対象から取り除いて実験した．

メソッドが出力された場合，元々プロジェクトに存在したメソッドと同一の処理をするものか目視で確認をした．

出力されるメソッドは次のように分類する．

加工無し再利用 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，任意の入力に対して自動生成対象のメソッドと同じ値を出力するメソッド

加工有り再利用 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，任意の入力に対して自動生成対象のメソッドと同じ値を出力するメソッド

オーバーフィット テストケースを通過することはできるが，ある入力に対して自動生成対象のメソッドとは異なる値を出力をするメソッド

この実験で自動生成に成功したメソッドとは，加工無し再利用，及び加工有り再利用の場合である．オーバーフィットは自動生成に失敗したものとして扱う．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り，null の場合の挙動がテストされていないメソッドに関しては，テストを作成した開発者がどのような挙動を期待していたのか不明なため，本論文では自動生成に成功したものとして扱った．それらのメソッドは加工の有無に応じて加工無し再利用か加工有り再利用のどちらかに分類した．

5.1.3 実験結果

実験結果を表 1 に示す．括弧で示された数字は null チェックの有無の違いがあるものの自動生成に成功したと扱った数である．以降では加工無し再利用，加工有り再利用，オーバーフィットについてのそれぞれの例を紹介する．

加工無し再利用の例

自動生成対象のメソッドは Algorithms[*5] 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 4 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．このメソッドを自動生成するにあたり，zxing[*6] 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．よって，この例は加工無し再利用に分類する．

```
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 4: 加工無し再利用の例

加工有り再利用の例

自動生成対象のメソッドは commons-lang 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，加工する際に再利用された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 5 に示す．このメソッドを自動生成するにあたり，fastjson[*7] 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 5(b) のソースコードから図 5(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た．再利用されたメソッドに加工が加えられているので，この例は加工有り再利用に分類する．図 5(a) と図 5(c) を比べると，簡潔なメソッドが出力された．このように入出力が一致していれば異なる実装のメソッドが出力される場合もあることがわかる．

オーバーフィットの例

自動生成の対象となるメソッドは commons-jxpath[*8] というプロジェクトに存在している equalStrings メソッドである．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

[*5] https://github.com/pedrovgs/Algorithms

[*6] https://github.com/zxing/zxing

[*7] https://github.com/alibaba/fastjson

[*8] https://github.com/apache/commons-jxpath

表 1: 実験 A の結果

自動生成対象のメソッド数	加工無し再利用の数	加工有り再利用の数	オーバーフィットの数
176	14(2)	2(0)	2

```
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 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);
}
```

(b) 加工する際に再利用されたメソッド

```
  String uncapitalize(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) 出力されたメソッド

図 5: 加工有り再利用の例

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 6 に示す．このメソッドを自動生成するにあたり，dubbo[*9]というプロジェクトの isEquals メソッドを再利用した．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 6(a) と図 6(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であることがわかる．

5.2 実験 B-1 について

5.2.1 DB の構築

表 2 に示されている 4 つのプロジェクトに対してそれぞれのメソッドが登録されている 4 つの DB を構築した．

5.2.2 自動生成対象のメソッド

表 2 の 4 つのプロジェクトのメソッドの内，命令カバレッジとブランチカバレッジがともに 100% であるものを自動生成の対象にした．カバレッジを用いる理由はすでに 5.1.2 小節で説明した通りである．またセッターやゲッターのように処理の内容が短いものはメソッドの自動生成という本手法のコンテキストとは合わないため，ステートメント数が 1 のメソッドは対象としていない．各プロジェクトの自動生成対象のメソッドの数は表 2 の通りである．

5.1.2 小節と同様にメソッドが出力された場合は元々プロジェクトに存在するメソッドと同様の処理をするメソッドか目視で確認をし，分類する．

[*9] http://dubbo.io

```
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 6: オーバーフィットの例

```
public boolean contains(final Object obj) {
    for (final Set<E> item : all) {
        if (item.contains(obj)) {
            return true;
        }
    }
    return false;
}
```

(a) 自動生成の対象となるメソッド

```
public boolean remove(final Object obj) {
    for (final Set<E> set : getSets()) {
        if (set.contains(obj)) {
            return set.remove(obj);
        }
    }
    return false;
}
```

(b) 加工する際に再利用されたメソッド

```
  public boolean contains(final Object obj) {
      for (final Set<E> set : getSets()) {
          if (set.contains(obj)) {
-             return set.remove(obj);
+             return true;
          }
      }
+     final Object[] result = new Object[size()];
      return false;
  }
```

(c) 出力されたメソッド

図 7: 加工有り再利用の例

5.2.3 実験結果

実験結果を表 2 に示す．この実験では自動生成対象のメソッドと出力されたメソッドに null チェックの有無が異なるものは出力されなかった．以降では加工有り再利用，オーバーフィットについてのそれぞれの例を紹介する．

加工有り再利用の例

自動生成対象のメソッドは commons-lang に存在している contains メソッドである．このメソッドは引数で与えられたオブジェクトがフィールド変数の配列に含まれている

か判定するメソッドである．このメソッドのソースコード，加工する際に再利用された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 7 に示す．このメソッドを自動生成するにあたり，同一プロジェクトの remove 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(b) から，図 7(c) で示された網かけの部分のコードが変更されて出力されている．再利用されたメソッドに加工が加えられているので，この例は加工有り再利用に分類する．

オーバーフィットの例

自動生成対象のメソッドは commons-lang に存在している removeStart 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 remove を引数で与えられた文字列 str の接頭から取り除くメソッドである．このメソッドのソースコード，加工する際に再利用された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の図 8 に示す．このメソッドを自動生成するにあたり，同一プロジェクトの removeEnds 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 8(b) から，図 8(c) で示された網かけの部分のコードが変更されて出力されている．しかし，str.startsWith(remomve) となるべきコードが，startsWithIgnoreCase(str, remove) となっている．startsWithIgnore メソッドは，startsWith メソッドの処理に，大文字小文字を区別しないという性質を加えたものである．そのため，出力されたメソッドは自動生成の対象となったメソッドとは異なる処理をするので，オーバーフィットに分類する．

5.3 実験 B-2 について

この実験では，用いた DB と自動生成の対象となるメソッドは 5.2 節で述べたものと同じである．

```
public static String removeStart(final String str, final String remove) {
    if (isEmpty(str) || isEmpty(remove)) {
        return str;
    }
    if (str.startsWith(remove)){
        return str.substring(remove.length());
    }
    return str;
}
```

(a) 自動生成の対象となるメソッド

```
public static String removeEnd(final String str, final String remove) {
    if (isEmpty(str) || isEmpty(remove)) {
        return str;
    }
    if (str.endsWith(remove)){
        return str.substring(0, (str.length() - remove.length()));
    }
    return str;
}
```

(b) 加工する際に再利用されたメソッド

```
  public static String removeStart(final String str, final String remove) {
      if (isEmpty(str) || isEmpty(remove)) {
          return str;
      }
-     if (str.endsWith(remove)){
          return str.substring(0, (str.length() - remove.length()));
-     }
+     if (startsWithIgnoreCase(str, remove)){
+         return str.substring(remove.length());
+     }
      return str;
  }
```

(c) 出力されたメソッド

図 8: オーバーフィットの例

5.3.1 実験結果

実験結果を表 3 に示す．この実験では自動生成対象のメソッドと出力されたメソッドに null チェックの有無が異なるものは出力されなかった．この実験で出力されたオーバーフィットの例を示す．

オーバーフィットの例

自動生成対象のメソッドは commons-text に存在している replaceAll メソッドである．このメソッドは引数で与えられた文字 search を引数で与えられた文字 replace に置換するメソッドである．このメソッドのソースコード，加工する際に再利用された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の図 9 に示す．このメソッドを自動生成するにあたり，同一プロジェクトの replaceFirst 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

表 2: 実験 B-1 の結果

プロジェクト名	プロジェクト内のメソッド数	自動生成対象のメソッド数	加工無し再利用	加工有り再利用	オーバーフィット
commons-text	98	66	0	5	2
commons-lang	512	415	0	10	8
commons-io	142	65	0	0	0
commons-collections	492	168	0	6	3

表 3: 実験 B-2 の結果

プロジェクト名	自動生成対象のメソッド数	加工無し再利用	加工有り再利用	オーバーフィット
commons-text	66	0	0	2
commons-lang	415	0	0	0
commons-io	65	0	0	0
commons-collections	168	0	0	0

```
public StrBuilder replaceAll(final char search, final char replace) {
    if (search != replace) {
        for (int i = 0; i < size; i ++) {
            if (buffer[i] == search) {
                buffer[i] = replace;
            }
        }
    }
    return this;
}
```

(a) 自動生成の対象となるメソッド

```
public StrBuilder replaceFirst(final char search, final char replace) {
    if (search != replace) {
        for (int i = 0; i < size; i ++) {
            if (buffer[i] == search) {
                buffer[i] = replace;
                break;
            }
        }
    }
    return this;
}
```

(b) 加工する際に再利用されたメソッド

```
public StrBuilder replaceAll(final char search, final char replace) {
    if (search != replace) {
        for (int i = 0; i < size; i ++) {
            if (buffer[i] == search) {
                buffer[i] = replace;
+               if (newLine != null)
                    break;
            }
        }
    }
    return this;
}
```

(c) 出力されたメソッド

図 9: オーバーフィットの例

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，Nopol によって加工が施された．図 9(b) から，図 8(c) で示された網かけの部分のコードが変更されて出力されている．しかし，出力されたソースコードは変数 newLine が null かどうかで break するタイミングが変わってしまう．そのため，出力されたメソッドは自動生成の対象となったメソッドとは異なる処理をするので，オーバーフィットに分類する．

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 実験 A に対する考察

実験 A で出力することに成功した自動生成対象のメソッドは apache commons に含まれるメソッドが多かった．一方で再利用されたメソッドに関してプロジェクトの偏りはなく，様々なプロジェクトのメソッドが再利用・加工されていた．このことから，メソッドを収集する際にプロジェクト間での優劣などは存在せず，一つでも多くのプロジェクトからメソッドを収集するべきであることがわかる．

これ以降では出力されたメソッドについて，加工無し再利用，加工有り再利用，オーバーフィットの 3 つに分けて考察する．

6.1.1 加工無し再利用に対する考察

加工無し再利用が多数出力された理由として，DB に登録したメソッド群と自動生成対象のメソッド群に同一の処理をするメソッドが含まれていることがあげられる．5.1 節で述べたように，DB に登録するメソッド群と自動生成対象のメソッド群にはいくつかの共通の特徴がある．この共通の特徴を持つメソッドを収集した結果，同一の処理をするメソッドがそれぞれのメソッド群に多く含まれてしまったと考えられる．つまり複数のプロジェクトにまたがったコードクローンが加工なし再利用として出力されていることが分かる．

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を自動生成することが有益である．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．加工無し再利用が出力されることは本論文の実験としては成功である．

6.1.2 加工有り再利用に対する考察

再利用されたメソッドが多く出力されたことに対して，加工有り再利用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 の加工能力が十分でなかった

本論文の提案手法では，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． 実験で出力されたものはいずれも 2, 3 行程度の加工であり，加工対象メソッドの処理を大きく書き換えるような加工はされていなかった． 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 の加工能力の問題がある．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 [7]．今後の GenProg の発展に応じて，加工有り再利用の出力数も増えることが考えられる．

ただし，加工無し再利用に比べて加工有り再利用の数が少ないこと自体は問題ではない．すでに加工無し再利用に対する考察で述べた通り，開発者の立場で考えると既存のメソッドから加工されたかどうかは問題ではない．

6.1.3 オーバーフィットに対する考察

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．実験をするにあたり，自動生成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 なメソッドも含めている．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，命令カバレッジとブランチカバレッジがともに 100%であるにも関わらず十分なテストケース

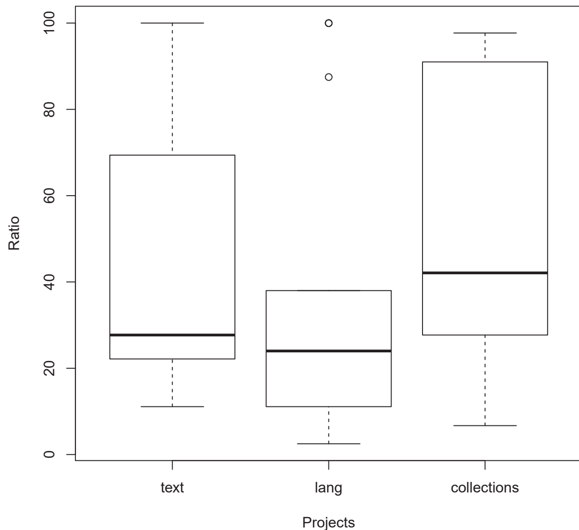

図 10: 自動生成に用いた加工対象メソッドの割合

が用意されず，オーバーフィットとなるメソッドを生成してしまったと考えられる．実験 A で出力されたオーバーフィットなメソッドは全て private なメソッドであった．

6.2 実験 B-1 に対する考察

実験 B-1 では，加工無し再利用の出力がまったく存在しなかった．これは 1 つのプロジェクト内に同じ処理をするメソッドが複数存在しないからである．正しく開発されているプロジェクトの場合，DRY の原則から 1 つのプロジェクト内に同じ処理をするメソッドは存在しないことが多い．そのため実験 B-1 では加工無し再利用は出力されなかったといえる．

また加工有り再利用の加工された行数は全て 1 行から 3 行程度の書き換えであった．このことから，GenProg によって再利用されるメソッドの処理が大きく変えられていないことがわかる．

6.2.1 加工対象メソッドの並び替え

3 章で述べたように DB から取り出されたメソッドに対して Step 2 で並び替えを行う．自動生成された各メソッドが，並び替えられた加工対象のメソッドのうち何番目を使用することで生成されたかを調べた．図 10 にその結果を箱ひげ図として示す．箱ひげ図における横軸は，対象プロジェクトのうち，自動生成されたメソッドがなかった commons-io 以外の 3 プロジェクトを示している．また縦軸は，加工対象のメソッド数に対する加工に用いたメソッド数の割合である．加工に用いたメソッド数はすなわち，3 章で述べた Step 3 を行った回数である．図から，3 つのプロジェクトにおいて中央値が 50%を下回っていることがわかる．このことから，メソッドのランク付けの手法として，メソッド名の類似度は有用であると考えられる．

6.3 実験 B-2 に対する考察

既存研究において，プログラム修正に関しては GenProg より Nopol の方が多くの欠陥を修正することができている [12]．しかし，この実験で Nopol はオーバーフィットなメソッドしか出力しなかった．また，出力されたオーバーフィットなメソッドは全て加工対象のメソッドに true になることのない if 文を追加しているものであった．つまり，true になることのない if 文を追加することで実行するべきでない文を避けている．これは GenProg では行の削除で実現することができる．このことから，メソッドの生成に関しては再利用に基づく手法である GenProg の方が有用であるといえる．

7. 実験の妥当性について留意する点

本論文では自動生成に成功したかどうかを目視で確認した．自動生成に失敗しているにも関わらず成功したものとして扱われていたり，自動生成に成功しているにも関わらず失敗したとして扱われていたりする可能性がある．

8. 既存研究との差異

本論文と類似した入出力をもつ手法として Reiss の研究 [13] が存在する．既存のメソッドに対していくつかのアプローチを施し，目的のメソッドを得る手法である．この手法ではあらかじめ設定したパターンの加工をしていくため，パターンから外れた加工をすることができない．例えばそのプロジェクト内のメソッドを用いた加工を施すことができない．提案手法の場合，GenProg を用いて同一プロジェクト内に存在するプログラムを挿入することで加工を施すので，より柔軟に加工を施すことができる．

9. あとがき

本論文では，既存のメソッドを再利用・加工することでメソッドの自動生成を行い，提案手法でメソッドが自動生成できるか実験を行った．その結果 37 個のメソッドの自動生成に成功した．

現時点では自動生成できたメソッドの割合は少なく，実用的な手法であるとはいえない．多くのメソッドを自動生成できるということは，開発者が大部分のコードを書く必要がなくなることを意味しており，現在の技術では実現が難しい．今後は，コードの自動生成技術が少しずつ発展し，開発者が徐々にコーディングから開放されていくだろう．本研究はコードの自動生成の初期段階の研究として重要であると著者らは考えている．

今後の課題として Reiss の研究の手法を提案手法に適用することが挙げられる．例えば引数と返り値の型の変換である．本論文では引数や返り値の継承関係などは解析せず，引数と返り値の型が全く同じメソッドのみを再利用している．Reiss の研究のように開発者が入力したシグネチャと

同じ型だけでなく，継承関係がある型もDBから取り出して加工の対象にすることで，より自動生成の成功率が上がることが予想される．このようにReissの研究を取り入れることで，より多くの加工をすることができ，実験の成功率も上がると予想される．

また，GenProgやNopol以外にも様々な自動プログラム修正の手法が存在する．どのような自動プログラム修正の手法がメソッドの自動生成に有用か比較実験も行なっていきたいと考えている．

10. 謝辞

本研究は，科学研究費補助金基盤研究(B)(課題番号：17H01725)の助成を得て行われた．

参考文献

[1] Solar-Lezama, A.: Program sketching, *International Journal on Software Tools for Technology Transfer*, Vol. 15, No. 5-6, pp. 475–495 (2013).

[2] Gvero, T. and Kuncak, V.: Synthesizing Java expressions from free-form queries, *Acm Sigplan Notices*, Vol. 50, No. 10, pp. 416–432 (2015).

[3] Gulwani, S., Harris, W. R. and Singh, R.: Spreadsheet data manipulation using examples, *Communications of the ACM*, Vol. 55, No. 8, pp. 97–105 (2012).

[4] 森畑明昌：プログラミングするプログラム-自動プログラム作成最前線，情報処理，Vol. 57, No. 6, pp. 544–549 (2016).

[5] Abreu, R., Zoeteweij, P., Golsteijn, R. and Van Gemund, A. J. C.: A Practical Evaluation of Spectrum-based Fault Localization, *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792 (2009).

[6] Koza, J. R.: *Genetic programming: on the programming of computers by means of natural selection*, Vol. 1, MIT press (1992).

[7] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Weimer, W.: 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*, ICSE '12, Piscataway, NJ, USA, IEEE Press, pp. 3–13 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2337223.2337225⟩ (2012).

[8] Barr, E. T., Brun, Y., Devanbu, P., Harman, M. and Sarro, F.: The Plastic Surgery Hypothesis, *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 306–317 (2014).

[9] Corazza, A., Martino, S. D., Maggio, V. and Scanniello, G.: Investigating the Use of Lexical Information for Software System Clustering, *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, Washington, DC, USA, IEEE Computer Society, pp. 35–44 (online), DOI: 10.1109/CSMR.2011.8 (2011).

[10] Higo, Y. and Kusumoto, S.: 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*, FSE 2014, New York, NY, USA, ACM, pp. 294–305 (online), DOI: 10.1145/2635868.2635886 (2014).

[11] Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D. and Monperrus, M.: Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs, *IEEE Trans. Softw. Eng.*, Vol. 43, No. 1, pp. 34–55 (online), DOI: 10.1109/TSE.2016.2560811 (2017).

[12] Martinez, M., Durieux, T., Sommerard, R., Xuan, J. and Monperrus, M.: Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset, *Empirical Software Engineering*, Vol. 22, No. 4, pp. 1936–1964 (2017).

[13] Reiss, S. P.: Semantics-based Code Search, *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Washington, DC, USA, IEEE Computer Society, pp. 243–253 (online), DOI: 10.1109/ICSE.2009.5070525 (2009).