修士学位論文

題目

シグネチャ情報と入出力情報を用いた **Java** メソッドの自動プログラミング手法

指導教員

楠本　真二　教授

報告者

下仲　健斗

平成 30 年 2 月 7 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 29 年度 修士学位論文

シグネチャ情報と入出力情報を用いた
Java メソッドの自動プログラミング手法

下仲　健斗

内容梗概

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．既存研究として，入出力例を与える手法が提案されている．ある入力と，それを与えたときに期待される出力とのペアを 1 つの入出力例とし，その入出力例を満たすようなプログラムを生成する手法である．また，自然言語を入力として与える手法も提案されている．プログラムを書くのではなく，目的のプログラムの振る舞いを自然言語で記述する．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．

本研究では，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（引数の型と返値の型，メソッド名）および入出力情報（引数の値と返値の組の集合）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，4 つのオープンソースプロジェクトに対して本手法を適用した．その結果，18 の Java メソッドを自動生成することに成功した．

主な用語

自動プログラミング，自動プログラム修正

目次

1 まえがき **1**

2 関連研究 **3**

2.1 プログラム合成 3

2.1.1 スケッチング 4

2.1.2 AnyCode 4

2.1.3 FlashFill 4

2.2 プログラム変換 5

2.2.1 Refazer 5

2.2.2 Code Carbon Copy 5

2.3 自動プログラム修正 6

2.3.1 GenProg 6

2.3.2 Nopol 6

2.4 ソースコード再利用 7

3 研究の動機 **9**

4 提案手法 **12**

5 実装 **14**

5.1 データベース 14

5.2 メソッドのランク付け 14

5.3 メソッドの加工 14

6 適用実験 **16**

6.1 実験手順 16

6.2 テストケースの調査 17

6.3 jGenProg を用いた場合の実験結果 17

6.4 Nopol を用いた場合の実験結果 22

7 考察 **25**

7.1 自動生成されたメソッド 25

7.2 オーバーフィッティングなメソッド 25

7.3 jGenProg と Nopol の比較 26

7.4 加工対象メソッドのランク付け 26

7.5 実行時間 26

8　妥当性の脅威 **27**
8.1　実験対象 27
8.2　自動プログラム修正ツール 27
8.3　自動生成されたメソッド 27

9　あとがき **28**

謝辞 **29**

参考文献 **30**

図目次

1 姓名連結の例 5
2 startsWith メソッド 9
3 endsWith メソッド 9
4 生成された startsWith メソッド 10
5 提案手法の概要 12
6 メソッド加工の処理の流れ 15
7 jGenProg によるオーバーフィッティングの例 1 18
8 jGenProg によるオーバーフィッティングの例 2 19
9 jGenProg による正しい自動生成の例 1 20
10 jGenProg による正しい自動生成の例 2 21
11 Nopol によるオーバーフィッティングの例 1 22
12 Nopol によるオーバーフィッティングの例 2 23
13 自動生成に用いた加工対象メソッドの割合 25
14 自動生成に要した時間 27

表 目 次

1　実験対象のメソッド 16
2　jGenProg を用いた場合の実験結果 17
3　Nopol を用いた場合の実験結果 24

1 まえがき

自動プログラミングとは，開発者が作成したい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．自動プログラミングは古くから研究されており，これまでいくつかの手法が提案されている．

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．

このように，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

本研究では，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．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，返値の型，およびメソッド名)
- 入出力情報 (引数の値と返値の組の集合)

Java 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，本研究では返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，ソフトウェア工学における最も有名な開発モデルである．本手法における入力は，ソフ

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

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追加の作業を行う必要はない．

提案手法を評価するために，4 つのオープンソースプロジェクトに対して実験を行った．実験では，プロジェクトからあるメソッドを除外し，残りのメソッドを加工することで，除外したメソッドの自動生成を試みた．この処理を合計 1,244 のメソッドに対して行った．実験の結果，18 の Java メソッドを自動生成することに成功した．

以降，2 章で関連研究について述べ，3 章では研究の動機について述べる．4 章で提案手法について説明し，5 章では提案手法の実装について説明する．6 章では，適用実験について述べ，7 章では実験に対する考察を述べる．8 章で妥当性の脅威について述べ，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　関連研究

2.1　プログラム合成

自動プログラミングの手法として，プログラム合成があり，近年盛んに研究されている [5][6]．プログラム合成では，開発者の意図を何らかの形で表現し，それを満たすプログラムを探索し，合成する．プログラム合成において考慮すべき点は，主に以下の 3 つである [5]．

1. 開発者の意図の表現
2. プログラムの探索空間
3. 探索手法

1. 開発者の意図の表現とは，目的のプログラム (開発者が作りたいプログラム) が満たすべき仕様を簡潔に表すことである．表し方の例としては，論理式，自然言語，入出力例などが挙げられる．Itzhaky らは，論理式を入力としたプログラム合成の手法を提案した [7]．論理式を用いる場合，開発者は論理式に関する知識が必要なため，自然言語や入出力例と比較すると開発者への負担が大きい．そこで，自然言語と論理式をマッピングする手法が提案されている [8]．自然言語の欠点としては，入力となる開発者の意図の表現があいまいになることである．入出力例を用いたプログラム合成では，どのような入出力例を用いれば仕様が十分に満たされるのか，という問題が存在し，開発者が判断するケースとプログラム合成器が判断するケース [9] がある．開発者が判断するケースでは，合成されたプログラムを開発者が見て，期待するプログラムでなければ新しい入出力例を追加する．プログラム合成器が判断するケースでは，与えられた入出力例を満たすプログラムが複数合成された場合，プログラム合成器が新しい入力を生成し，開発者が出力を補完するというものである．

2. プログラムの探索空間とは，どのようなプログラムを合成の対象とするかである．ソースコードを合成する場合，ソースコードの制約を設けることがある．例えば，演算子の制約，制御構造の制約などである．演算子の制約では，算術演算子を用いるのか，ビット演算子を用いるのかなどを選択する．制御構造の制約では，ループの有無，プログラム文の個数の制限などがある．また，ソースコードの合成以外にも，文法 (正規表現や文脈自由文法) や論理式を合成する場合もある．

3. 探索手法としては，力任せ法，機械学習に基づく手法，論理的推論に基づく手法などがある．力任せ法は，探索空間のプログラムが入力を満たすかどうかしらみつぶしに探索する手法である．力任せ法は最も単純な探索手法であり，規模の小さい関数型言語を対象としたプログラム合成によく用いられる [10]．機械学習に基づく手法では，確率伝搬法 [11] を用いる場合，遺伝的プログラミング [12] を用いる場合などがある．論理的推論に基づく手法とは，プログラム合成を SMT 問題に帰着させ，SMT ソルバによって問題を解く手法である．SMT 問題は NP-完全な決定問題であり，多項式時間での解法が存在せず，網羅的な探索が必要である [13]．SMT ソルバとは，SMT 問題を高速に解くためのツールであり，Z3[14] や，CVC4[15] などが知られている．

ここで，プログラム合成を用いたツールについて説明する．

2.1.1　スケッチング

Armando は，スケッチングという手法を提案した [1]．スケッチングでは，プログラムの概形を与え，プログラムの細部を自動で埋めてくれる．例として，ビット列の最下位の 1 のビットを抽出するプログラムを考える．例えば，01100100 を入力として与えると 00000100 が出力される．このプログラムをビット演算を用いて実現したいが，開発者は部分的にしか書けず，以下のようなプログラムを書いたとする．

```
y = ~(x + ??) & (x + ??)
```

??の部分はスケッチングによって埋めてほしい部分である．スケッチングは SMT ソルバを用いることで??の部分を以下のように埋める．

```
y= ~(x + 0xFFFFFFFF) & (x + 0)
```

2.1.2　AnyCode

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．例えば，開発者が fname という名前のファイルのバックアップとして bname という名前のファイルを作りたいとする．そこで以下のような自然言語を AnyCode の入力として与える．

```
copy file fname to bname
```

すると，以下のような関数呼び出しの候補が提示される．

- FileUtil.copyFile(new File(fname), new File(bname))
- FileUtil.copyFile(new File(bname), new File(fname))
- FileUtil.copyFileToDirectory(new File(fname), new File(bname))

開発者は，一番上の候補を選択すれば，目的の処理を実現することができる．

2.1.3　FlashFill

Galuwani らは，文字列操作の自動化ツールである FlashFill 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．図 1 に姓名連結の例を示す．エクセルの表の A 列に人の苗字，B 列に人の名前が入力されており，開発者は C 列に A 列と B 列を連結させたもの，つまり姓名連結を入力したい．開発者は C 列の 1 行目だけ手動で入力し，FlashFill を実行すると，残りの行が自動で入力される．

図 1: 姓名連結の例

2.2　プログラム変換

プログラム合成と同様に，プログラム変換についても近年盛んに研究されている．プログラム変換とは，すでに存在しているプログラムに対し，何らかの変更を自動で行う技術のことである．主にリファクタリングやプログラム修正などに用いられる．

2.2.1　Refazer

ソースコードのリファクタリングにおいて，似たような変更を複数箇所に行うことは煩雑な作業であり，バグの混入にもつながる．Rolim らは，変更履歴を用いて，似たような変更を自動で行うツール，Refazer を開発した [16]．Refazer の入力となるのは，ソースコードをどのように変更したかを表す，変更履歴である．Refazer は変更履歴を解析し，その変更が適用可能な箇所を特定する．開発者は，Refazer によって特定された箇所に変更を加える必要があると判断したら，Refazer はその箇所に対し適切な変更を加える．既存研究において，Refazer は 3 つのオープンソースソフトウェアに対して適用され，56 個の変更中 40 個の変更は，開発者と同じ変更であった．さらに 56 個中 7 個に関しては，開発者が見逃していた変更であった．

2.2.2　Code Carbon Copy

プログラムを実装する際，似たようなプログラムを探し，コピーアンドペーストで機能移植をするということは頻繁に行われる．しかし，コピーしたプログラムは必ずしもそのまま利用できるわけではなく，データ構造や変数名の違いなどにより，ある程度の修正が必要である．Douskos らは，そのような修正を自動化して，プログラムの機能移植を支援するツール，CodeCarbonCopy(以下 CCC) を開発した [17]．CCC では，移植するプログラムの探索，および移植するプログラムの挿入先の決定は開発者が行い，その後の挿入作業を CCC が自動で行う．具体的には，変数のマッピング，配列型変数のデータ構造のマッピング，および無意味な文の削除である．CCC は関数単位で機能移植を行うため，移植する関数内でグローバル変数を用いている場合，そのまま挿入先プログラムにコピーはできない．そこで，CCC はグローバル変数を関数の引数として与え，挿入先プログラム内でもそ

の変数を参照できるようにする．また，CCC は主に画像や動画処理に関するプログラムを対象としており，そのようなプログラムはデータを配列に格納する傾向にある．配列への格納方法 (何番目にどのデータが格納されているか) がプログラムにより異なるため，CCC は配列のマッピングを行っている．さらに，CCC はプログラム変換を行った後，目的の処理には関係のない文を削除する．CCC は 6 つのオープンソースソフトウェアに対して適用され，計 8 つの機能移植のうち 7 つの機能移植に成功した．

2.3 自動プログラム修正

自動プログラム修正とは，プログラム中の欠陥箇所を特定し，その欠陥を修正する，いわゆるデバッグと呼ばれる作業を自動化することである．欠陥箇所の特定は，テスト集合を用いて行われる．具体的には，プログラムのテストを実行して通過しないテストが存在する場合に，通過テストと未通過テストの割合を用いて欠陥である可能性の高さを算出する．

2.3.1 GenProg

GenProg は Weimer らが開発した自動プログラム修正ツールである [18]．欠陥箇所の限局の手法として GenProg 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を用いている [19]．また，欠陥箇所の修正の手法として GenProg は，自動プログラム修正の手法の 1 つである再利用に基づく手法を採用している．この手法では，修正対象プログラムに存在するプログラム文を用いて欠陥箇所を変更したプログラムを生成する．GenProg は遺伝的プログラミングに基づき，この操作を変更したプログラムが全てのテストケースを通過するまで繰り返し行う．プログラムの変更は，プログラム文単位で行われ，具体的な処理は以下の 3 つである．

挿入 欠陥箇所にプログラム文の挿入を行う処理

削除 欠陥箇所を削除する処理

置換 削除処理と挿入処理を同時に行う処理

既存研究において，GenProg は 8 つのオープンソースソフトウェアに対して適用され，105 個中 55 個の欠陥の修正に成功している [20]．しかし，GenProg は既存のプログラム文を用いて変更を行うため，プログラム中に存在しない文が必要な欠陥は修正できないことや [21]，修正に要する時間が膨大であるといった課題もある [20]．

2.3.2 Nopol

Nopol は if 文の追加および if 文の条件式の変更により修正を行うツールである [22]．Nopol の修正手法は以下の 2 段階から構成される．

- 満たすべき制約の収集

- プログラム合成

満たすべき制約の収集では，プログラムの制御が欠陥の所在に達した時に，スコープ内の変数が満たさなければならない制約をテストの実行により収集する．プログラム合成では，収集した変数の制約を満たすように if 文を追加したり if 文の条件式を変更したりする．制約を満たすような式を生成する際，Nopol は制約を SMT 問題に変換し，SMT ソルバによって問題を解く．

Martinez らは，Defects4J[23] を用いて GenProg と Nopol の比較実験を行った [24]．Defects4J とは Java で開発された 5 つのオープンソースソフトウェア (JFreeChart，Closure Compiler，Commons Lang，Commons Math，Joda time) の開発過程で発生した 357 個の欠陥を収集したものである．実験対象は，Closure Compiler 以外の 224 個の欠陥である．Closure Compiler は，JUnit を用いたテストケースを扱っていないため対象からは除外されている．実験の結果，GenProg は 27 個の欠陥を修正し，Nopol は 35 個の欠陥を修正した．よって，自動プログラム修正の精度においては，Nopol の方が高い結果となっている．

2.4 ソースコード再利用

ソフトウェア開発において，効率的にコードを書くための方法として注目されるのが，ソースコードの再利用である．既存のコードの中から，目的のコードと似通ったものを探し，再利用することで，効率的に開発することができる．しかし，再利用元となるオープンソースソフトウェアの数は膨大であり，その中から目的のコードを探索するのは容易ではない．そこで近年では，GitHub[2] や Krugle[3] などのように，キーワードによるソースコード検索が利用されている．

しかし，キーワードのみによるソースコード検索では，開発者の意図に合致するソースコードが効率よく見つからず，しばしば自身でソースコードを書く以上の労力が必要とされる [25][26]．効率よく目的のソースコードを検索するために，様々な手法が提案されている．1990 年代に提案された手法の中には，シグネチャを用いる手法 [27]，関係式を用いる手法 [28]，パターンマッチングを用いる手法 [29] などがある．

これらの既存手法には，条件が弱すぎて再利用候補が膨大になるという問題や，条件が強すぎて再利用候補がほとんどないという問題などが存在する．そこで近年では，テストケースを用いたソースコード探索手法が提案されている．Lemos らは，テストケースを用いて大規模なリポジトリからソースコードを探索および再利用するツール，CodeGenie を開発した [30]．開発者は検索のクエリとしてテストケースを作成する．CodeGenie は，そのテストケースを満たすソースコードを出力する．テストケースを用いた探索手法は，ウェブサービスの分野にも適用されている [31]．

上述した手法は，ソースコードを探索するのみで，ソースコードの書き換えは行わない．Reiss は，ソースコードの探索を行った後にソースコードの書き換えも行うツール，S^6 を開発した [26]．S^6 のソースコード探索は，キーワードとシグネチャ情報を用いた静的な探索とテストケースを用いた動

[2]https://github.com/
[3]http://www.krugle.com/

的な探索に分けられる．静的な探索では，まずキーワード検索で再利用候補を抽出する．次に，シグネチャ情報を用いて再利用候補の絞り込みを行う．動的な探索では，上記の静的探索で得られた再利用候補に対して，テストケースを実行する．そのテストケースを通過したメソッドが出力される．ただし，テストケースを通過しなかったメソッドに関しては，候補から除くのではなく，テストケースを通過するようにソースコードの書き換えを行う．書き換え処理の内容としては，シグネチャ情報を変更する，メソッド内の処理を部分的に抽出する，新たに引数を追加する，などである．このように S^6 は，メソッドの外部的構造の変更を行っているが，本研究の手法のように内部的振る舞いの変更は行っていない．

```
1 public boolean startsWith(final String str){
2     if (str == null) {
3         return false;
4     }
5     final int len = str.length();
6     if (len == 0) {
7         return true;
8     }
9     if (len > (size)) {
10        return false;
11    }
12    for (int i = 0; i < len; i++) {
13        if ((buffer[i]) != (str.charAt(i))) {
14            return false;
15        }
16    }
17    return true;
18 }
```

図 2: startsWith メソッド

```
1 public boolean endsWith(final String str){
2     if (str == null){
3         return false;
4     }
5     final int len = str.length();
6     if (len == 0) {
7         return true;
8     }
9     if (len > (size)){
10        return false;
11    }
12    int pos = size - len;
13    for (int i = 0; i < len; i++ , pos++){
14        if ((buffer[pos]) != (str.charAt(i))){
15            return false;
16        }
17    }
18    return true;
19 }
```

図 3: endsWith メソッド

3 研究の動機

オープンソースプロジェクトである apache-commons-text(以下 commons-text) 内に，処理内容が類似した 2 つのメソッド，startsWith メソッドと endsWith メソッドが存在する．startsWith メソッドは，

```
1 public boolean endsWith(final String str){
2     if (str == null){
3         return false;
4     }
5     final int len = str.length();
6     if (len == 0) {
7         return true;
8     }
9     if (len > (size)){
10        return false;
11    }
12-    int pos = size - len;
12+    int post = 0;
13    for (int i = 0; i < len; i++ , pos++){
14        if ((buffer[pos]) != (str.charAt(i))){
15            return false;
16        }
17    }
18    return true;
19 }
```

図 4: 生成された startsWith メソッド

ある文字列 (文字列 1 とする) が入力となる文字列 (文字列 2 とする) で始まるかどうか判定するメソッドであり，endsWith メソッドは文字列 1 が文字列 2 で終わるかどうか判定するメソッドである．図 2 に startsWith メソッドのソースコード，図 3 に endsWith メソッドのソースコードを示す．図 2 および図 3 から，処理内容だけでなく，ソースコード自体も似通ったものであることがわかる．

ここで，startsWith メソッドが commons-text 内に存在していない状況を想定する．開発者は，文字列 1 が文字列 2 で始まるかどうか判定するようなメソッドを作成したい．もし開発者が，endsWith メソッドがプロジェクト内に既に存在していることを認識しているならば，コピーアンドペーストを行い，既存コードを少し書き換えることで startsWith メソッドを実装することができる．しかし，endsWith メソッドの存在を開発者が認識していない場合，上記の方法をとることはできない．

そこで本研究では，メソッドの仕様を入力として与えることで，目的のメソッドを自動生成する手法を提案する．上述した状況において，startsWith メソッドを生成したい場合，提案手法の入力におけるシグネチャ情報を以下のように与える．

- メソッド名：startsWith
- 引数の型：String
- 返値の型：boolean

また，入出力情報を以下のように与える．

入出力例 1 文字列 1：abcd，文字列 2：ab，出力：true

入出力例 2 文字列 1：abcd，文字列 2：abcd，出力：true

入出力例 3 文字列 1：abcd，文字列 2：cd，出力：false

図 4 に，提案手法を用いて得られた startsWith メソッドを示す．生成されたコードでは，12 行目で文の置換が行われていることがわかる．図 2 の startsWith メソッドでは，12 行目から始まる for 文において，buffer 配列を 0 番目から len 番目までチェックする処理となっている．図 3 の endsWith メソッドでは，13 行目から始まる for 文において，buffer 配列を pos 番目から len 番目までチェックする処理となっている．図 4 の生成コードでは，12 行目の代入文により，buffer 配列を 0 番目から len 番目までチェックする処理となっているため，startsWith メソッドが正しく生成されていることがわかる．このように，処理内容が類似したメソッドを加工することによって，自動生成が可能であることがわかる．

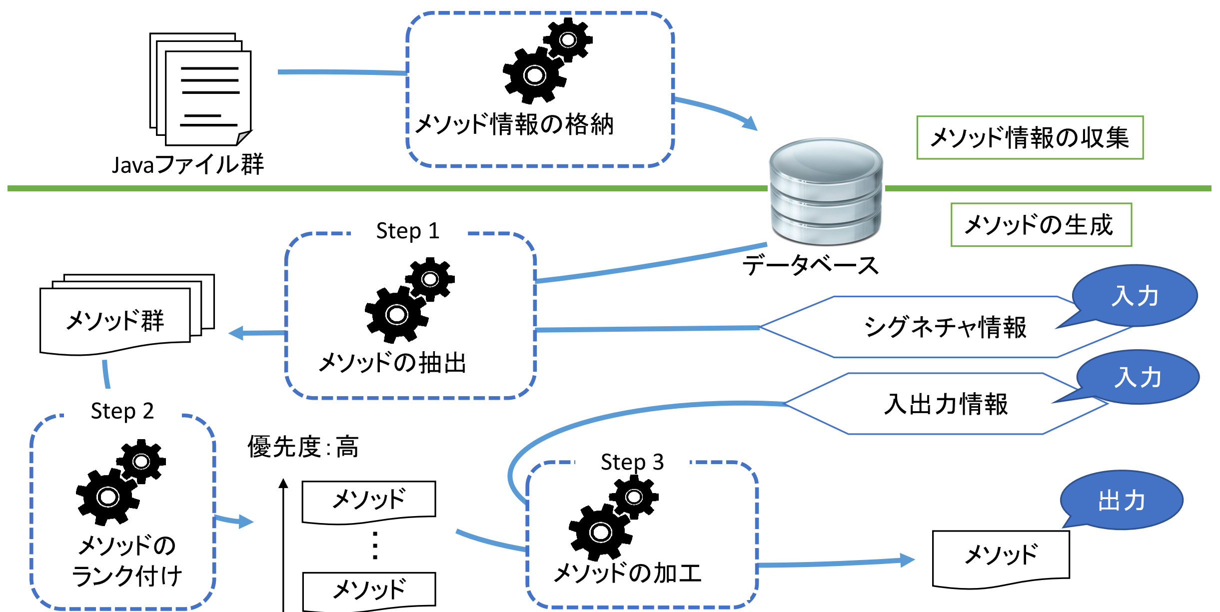

図 5: 提案手法の概要

4　提案手法

本研究では，Java メソッドの仕様からソースコードを自動生成する手法を提案する．提案手法の概要を図 5 に示す．提案手法の入力は Java メソッドのシグネチャ情報，およびテストケースである．出力は，仕様を満たす Java メソッドである．

提案手法は以下の 2 つの工程から構成される．

- メソッド情報の収集
- メソッドの生成

メソッド情報の収集では，プロジェクト内の Java ファイル群を解析し，各メソッドのシグネチャ情報をデータベースに格納する．格納したメソッドごとのシグネチャ情報は，メソッドの生成工程において利用される．つまり，自動生成したいメソッドの元となりうるメソッドである．

メソッドの生成工程は，次の 3 ステップから構成される．

Step 1　メソッドの抽出

入力されたシグネチャにおける引数と返値の型と一致するメソッドを，あらかじめ用意しておいたデータベースから抽出する．また，引数の順序に関しても，一致している必要がある．

Step 2 メソッドのランク付け

Step 1 にて抽出したメソッド群を優先度が高い順に並び替える．これは，抽出したメソッドの数が多い場合に，ランダムに選択すると自動生成に必要な時間が膨大になる可能性があるからである．したがって，開発者の意図に合致する可能性が高いメソッドを優先的に用いることが望ましい．本手法では，入力されたシグネチャにおけるメソッド名との類似度を優先度の基準とする．

Step 3 メソッドの加工

優先度が高いメソッドから順に，入力されたテストケースを満たすまで抽出したメソッドを加工する．本手法における加工とは，シグネチャは変えずに，メソッド内のコードに変更を加えることである．ただし，メソッドの加工後にもコンパイルが通るような変更である必要がある．

5 実装

提案手法の実装について述べる．ただし，対象とするプロジェクトは以下の条件を満たすものである．

- Java で開発されている
- JUnit を用いたテストケースが存在する

提案手法の実装は，以下の 3 つの要素からなる．

- メソッド情報を格納するデータベース
- メソッドのランク付け
- メソッドの加工

以降，3 つの実装方法について述べる．

5.1 データベース

データベースとして SQLite[32] を用いた．データベースに格納する属性は，引数と返値の型，メソッド名，ファイルパス，クラス名，プロジェクト名，メソッドの開始行，ソースコード，である．

5.2 メソッドのランク付け

メソッドのランク付けの指標として，メソッド名における類似度を用いる．これは，メソッド名が類似していると，メソッドの処理内容も類似している可能性が高いという先行研究の調査結果に基づいている [33][34]．入力されたメソッド名と，抽出されたメソッド群のメソッド名との類似度をレーベンシュタイン距離を用いて算出する．その値が小さいほど，優先度が高いと判断する．

5.3 メソッドの加工

メソッドの加工には，2 つの自動プログラム修正ツールを用いた．1 つ目は，GenProg[18] を Java で再実装した jGenProg である．GenProg は元々OCaml で開発されているが，Java で再実装された jGenProg[4] が公開されている．Martinez らは，3 つの自動プログラム修正ツール (GenProg, Kali[35], MutRepair[36]) を Java で再実装し，Astor というツールにまとめている [37]．本研究では Astor に内包されている jGenProg を用いる．2 つ目は Nopol を用いる．Nopol も，Astor を開発した Matinez らによって開発された自動プログラム修正ツールであり，Java で実装されている [5]．本研究では，jGenProg を用いたメソッドの加工と Nopol を用いたメソッドの加工の 2 種類を実装する．図 6 に具体的なメ

[4]https://goo.gl/9HzZbE
[5]https://goo.gl/LMhbTW

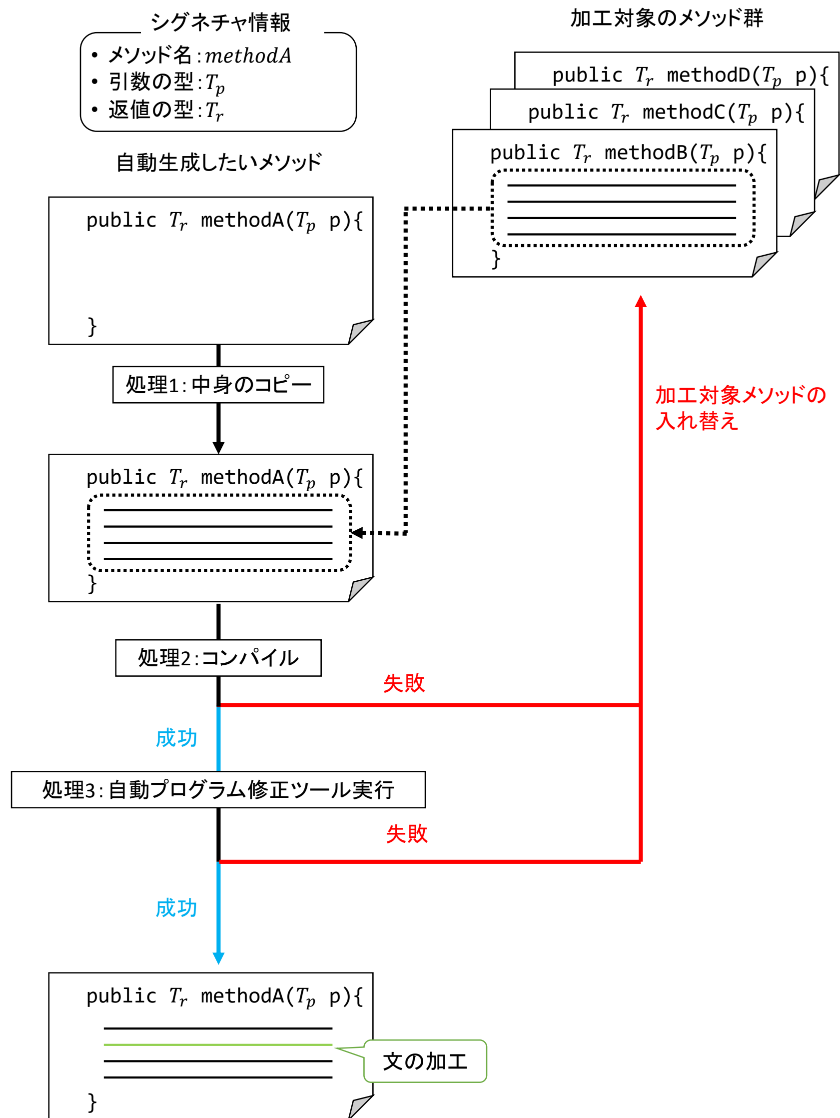

図 6: メソッド加工の処理の流れ

ソッドの加工の流れを示す．まず，自動生成したいメソッドと，引数と返値の型が同じメソッド (加工対象メソッドと呼ぶ) を取得する．また，自動生成したいメソッドとして，シグネチャ情報のみを反映させた，中身が空のメソッドを作る．その後の処理として，以下の 3 つに分かれる．

処理 1 加工対象メソッドの中身を自動生成したいメソッドにコピーする．

処理 2 コンパイルが成功すれば処理 3 に移行し，コンパイルが失敗すれば別の加工対象メソッドを用いて処理 1 に戻る．

処理 3 自動プログラム修正ツールを実行し，テストケースをすべて通過するメソッドが生成されれば，それを出力する．生成されなければ，別の加工対象メソッドを用いて処理 1 に戻る．

6　適用実験

本章では，4 つ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 章で述べたようなメソッドの生成が，プロジェクト全体でどの程度成功するのか調査することである．

6.1　実験手順

実験対象のプロジェクトから，自動生成したいメソッドを 1 つ選択する．その後，5.3 節で述べた処理を行い，自動生成を試みる．これを，プロジェクト内に存在する全ての実験対象メソッドに対して行う．

実験には，以下の 4 つのオープンソースプロジェクトを用いた．

- apache-commons-text
- apache-commons-lang
- apache-commons-io
- apache-commons-collections

上記のプロジェクトを選んだ理由は，Java で開発されており，JUnit を用いたテストケースも存在するからである．また，プロジェクト内に存在するメソッドのうち，実験対象となるのは以下の条件を満たすメソッドである．

- 加工対象メソッドが 1 つ以上存在
- 命令カバレッジとブランチカバレッジがともに 100%
- ステートメント数が 2 以上
- JUnit を用いたテストケースが存在

表 1: 実験対象のメソッド

プロジェクト名	実験対象メソッド数	加工可能なメソッド数
commons-text	98	66
commons-lang	512	415
commons-io	142	65
commons-collections	492	168

実験対象のメソッド数と加工可能なメソッド数を表 1 に示す．加工可能なメソッド数とは，実験対象メソッドのうち，5.3 節で述べた処理 2 において，コンパイル可能な加工対象メソッドが 1 つ以上存在するメソッドの数である．

カバレッジを用いる理由は 6.2 節で述べる．ステートメント数が 2 以上のメソッドを用いる理由は，ステートメント数が 1 のメソッドはセッターやゲッターであり，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からである．テストケースが存在しないと，提案手法が適用できないためテストケースが必要となる．また，提案手法における Step3 において，jGenProg と Nopol のタイムアウト時間を 30 分とした．また，jGenProg は，再利用するプログラム文のスコープを，プロジェクト，パッケージ，クラスの 3 つから選択可能である．本実験ではスコープをクラスとした．

6.2　テストケースの調査

プロジェクト内に存在するメソッドに対して提案手法を適用するためには，テストケースが必要である．一般的に，テストメソッドには，テスト対象のメソッド名の接頭に’test’ という文字列を付加するが，そうでない場合もある．したがって，テストメソッドの名前だけではテストケースが存在するかどうかの判別が困難である．そこで，各メソッドに対応するテストケースが存在するか調査するために，カバレッジを算出した．カバレッジの算出には，Eclipse のプラグインである EclEmma を用いた [38]．このツールでは，メソッドごとの命令カバレッジおよびブランチカバレッジを算出することができる．算出したカバレッジの値が高いものを，実験対象とする．

6.3　jGenProg を用いた場合の実験結果

メソッドの加工に jGenProg を用いた実験結果を表 2 に示す．自動生成されたメソッド数とは，全てのテストケースを通過するソースコードを生成できたメソッドの数である．正しく自動生成されたメソッド数とは，自動生成されたメソッドからオーバーフィッティングのメソッドを取り除いたものの数である．オーバーフィッティングとは，入力として与えられたテストケースは満たすが，それ以外は満たさない状態のことである．ただし，自動生成されたメソッドがオーバーフィッティングかどうかの判断は，目視確認によるものである．実験の結果，jGenProg により 31 個のメソッドが自動生成され，そのうち 18 個が正しい自動生成であった．以下，オーバーフィッティングであったメソッ

表 2: jGenProg を用いた場合の実験結果

プロジェクト名	自動生成されたメソッド数	正しく自動生成されたメソッド数
commons-text	7	5
commons-lang	18	10
commons-io	0	0
commons-collections	6	3

自動生成したいメソッド

```
1 public static String removeStart(final String str, final String remove) {
2     if (isEmpty(str) || isEmpty(remove)) {
3         return str;
4     }
5     if (str.startsWith(remove)){
6         return str.substring(remove.length());
7     }
8     return str;
9 }
```

加工対象メソッド

```
1 public static String removeEnd(final String str, final String remove) {
2     if (isEmpty(str) || isEmpty(remove)) {
3         return str;
4     }
5     if (str.endsWith(remove) {
6         return str.substring(0, (str.length() - remove.length()));
7     }
8     return str;
9 }
```

生成されたメソッド

```
1 public static String removeStart(final String str, final String remove) {
2     if (isEmpty(str) || isEmpty(remove)) {
3         return str;
4     }
5-    if (str.endsWith(remove) {
6-        return str.substring(0, (str.length() - remove.length()));
5+    if (startsWithIgnoreCase(str, remove)) {
6+        return str.substring(remove.length());
7     }
8     return str;
9 }
```

図 7: jGenProg によるオーバーフィッティングの例 1

ド，正しく自動生成されたメソッドの例を述べる．

オーバーフィッティングの例

図 7 は，common-lang における StringUtils クラスの removeStart メソッドを示している．行番号の横の’-’ は，削除された加工対象メソッドのコードを表す．’+’ は，既存のコードが挿入されたことを表す．removeStart メソッドは，文字列 remove を文字列 str の接頭から取り除くメソッドである．加工対象メソッドは同クラスの removeEnd メソッドであり，文字列 str の接尾から文字列 remove を取り除くメソッドである．加工処理としては，6 行目の if 文の条件式である．生成されたメソッドでは，

自動生成したいメソッド

```
1 public E peek() {
2     fill();
3     return exhausted ? null : slot;
4 }
```

```
1 private void fill(){
2    if (exhausted || slotFilled){
3        return;
4    }
5    if (iterator.hasNext()){
6        slot = iterator.next();
7        slotFilled = true;
8    } else {
9          exhausted = true;
10         slot = null;
11         slotFilled = false;
12   }
13 }
```

加工対象メソッド

```
1 public E element() {
2     fill();
3     if (exhausted) {
4         throw new NoSuchElementException();
5     }
6     return slot;
7 }
```

生成されたメソッド

```
1 public E peek() {
2     fill();
3     if (exhausted) {
4-          throw new NoSuchElementException();
4     }
5     return slot;
6 }
```

図 8: jGenProg によるオーバーフィッティングの例 2

6 行目で startsWithIgnoreCase メソッドを呼び出している．このメソッドは，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．removeStart メソッドに対する既存テストケースでは，大文字小文字の区別を考慮していないので，オーバーフィッティングとなっている．図 8 に，commons-collections における PeekingIterator クラスの peek メソッドを示す．peek メソッドは，クラスのフィールド変数であるイテレータの次の要素を返すメソッドである．次の要素がなければ null を返す．peek メソッド内で呼び出している fill メソッドは，同クラス内に存在しているメソッドである．加工対象メソッドは同クラスの element メソッドである．これは，peek メソッドと

自動生成したいメソッド

```
1 public static boolean isBlank(final CharSequence cs) {
2     int strLen;
3     if (cs == null || (strLen = cs.length()) == 0) {
4         return true;
5     }
6     for (int i = 0; i < strLen; i++) {
7         if (!isWhitespace(cs.charAt(i))) {
8             return false;
9         }
10    }
11    return true;
12 }
```

加工対象メソッド

```
1 public static boolean isWhiteSpace(final CharSequence  cs) {
2     if (isEmpty(cs))
3         return false;
4     final int sz = cs.length();
5     for (int i = 0; i < sz; i++) {
6         if (!isWhitespace(cs.charAt(i))))
7             return false;
8     }
9     return true;
10 }
```

生成されたメソッド

```
1 public static boolean isBlank(final CharSequence cs) {
2     if (isEmpty(cs))
3-        return false;
3+        return true;
4     final int sz = cs.length();
5     for (int i = 0; i < sz; i++) {
6         if (!isWhitespace(cs.charAt(i))))
7             return false;
8     }
9     return true;
10 }
```

図 9: jGenProg による正しい自動生成の例 1

同様に，イテレータの次の要素を返すメソッドであるが，次の要素がなければ例外を投げる．生成されたメソッドでは，4 行目の throw 文が削除されているため，次の要素がなければ何もしないという処理になっている．したがって，4 行目が文の削除ではなく，null を返す return 文との置換であれば正しい自動生成といえる．

正しく自動生成された例

次に図 9 は commons-lang における StringUtils クラスの isBlank メソッドを示している．isBlank メ

自動生成したいメソッド

```
1 public boolean contains(final Object obj) {
2     for (final Set<E> item : all) {
3         if (item.contains(obj)) {
4             return true;
5         }
6     }
7     return false;
8 }
```

加工対象メソッド

```
1 public boolean remove(final Object obj) {
2     for (final Set<E> set : getSets()) {
3         if (set.contains(obj)) {
4             return set.remove(obj);
5         }
6     }
7     return false;
8 }
```

生成されたメソッド

```
1 public boolean contains(final Object obj) {
2     for (final Set<E> set : getSets()) {
3         if (set.contains(obj)) {
4-            return set.remove(obj);
4+            return true;
5         }
6     }
7+    final Object[] result = new Object[size()];
8     return false;
9 }
```

図 10: jGenProg による正しい自動生成の例 2

ソッドは，与えられた文字列が空白，もしくは null かどうか判定するメソッドである．加工対象メソッドは同クラスの isWhiteSpace メソッドであり，与えられた文字列が空白かどうか判定するメソッドである．生成されたメソッドにおける 2 行目の if 文の条件式である isEmpty メソッドは自動生成したいメソッドの 3 行目の条件式と等価なメソッドである．したがって，生成されたメソッドは元のメソッドと等価な処理内容となっており，正しい自動生成といえる．また，生成されたメソッドにおける if 文の条件式がメソッド呼び出しであることは，元のコードよりも可読性が高いといえる．

図 10 に，commons-collections における CompositeSet クラスの contains メソッドを示す．contains メソッドは，List 型であるフィールド変数 all の中に，引数の obj が含まれているか判定するメソッドである．加工対象メソッドは同クラスの remove メソッドであり，contains メソッドと同様に all の中に obj が含まれているか判定し，含まれていたら削除するメソッドである．生成されたメソッドにおける 2 行目の getSets メソッドは all を返すメソッドである．4 行目が return true に置換されているため，元のメソッドと等価な処理になっていることがわかる．

自動生成したいメソッド

```
1 public StrBuilder appendNull() {
2     if (nullText == null)  {
3         return this;
4     }
5     return append(nullText);
6 }
```

加工対象メソッド

```
1 public StrBuilder appendNewLine() {
2     if (newLine == null) {
3          append(System.lineSeparator());
4          return this;
5     }
6      return append(newLine);
7 }
```

生成されたメソッド

```
1 public StrBuilder appendNull() {
2     if (newLine == null) {
3+        if (newLine != null) {
4              append(System.lineSeparator());
5+         }
6          return this;
7      }
8      return append(newLine);
9 }
```

図 11: Nopol によるオーバーフィッティングの例 1

6.4 Nopol を用いた場合の実験結果

メソッドの加工に Nopol を用いた場合の実験結果を表 3 に示す．Nopol により自動生成されたメソッドは計 2 個であり，どちらもオーバーフィッティングであった．以下，生成されたメソッドについて述べる．図 11 は commons-text における StrBuilder クラスの appendNull メソッドを示している．StrBuilder クラスは文字列を扱うクラスであり，char 型配列のフィールド変数 buffer の中に文字列を格納する．appendNull メソッドでは，開発者が指定した null 文字 (フィールド変数である nullText) を buffer に追加するメソッドである．ただし，nullText が null であれば buffer には何も追加しない．Nopol は appendNewLine メソッドを加工し，コードを生成した．appendNewLine メソッドは，改行文字を buffer に追加するメソッドであり，改行文字 (フィールド変数である newLine) は開発者が指定できる．ただし newLine が null ならば System クラスの改行文字が追加される．生成コードでは 3 行目に if 文が追加されており，条件式は 2 行目の条件式の否定となっている．ゆえに，4 行目の文が実行されることはないため，4 行目の文が削除された処理と等価であるといえる．テストケースでは nullText および newLine が null であったため，生成コードの 6 行目の文しか実行されず，結果

自動生成したいメソッド

```
1 public StrBuilder replaceAll(final char search, final
  char replace) {
2         if (search != replace) {
3             for (int i = 0; i < size; i++) {
4                 if (buffer[i] == search) {
5                     buffer[i] = replace;
6                 }
7             }
8         }
9         return this;
10    }
```

加工対象メソッド

```
1 public StrBuilder replaceFirst(final char search,
  final char replace) {
2         if (search != replace) {
3             for (int i = 0; i < size; i++) {
4                 if (buffer[i] == search) {
5                     buffer[i] = replace;
6                     break;
7                 }
8             }
9         }
10         return this;
11    }
```

生成されたメソッド

```
1 public StrBuilder replaceAll(final char search,
  final char replace) {
2         if (search != replace) {
3             for (int i = 0; i < size; i++) {
4                 if (buffer[i] == search) {
5                     buffer[i] = replace;
6+                    if(newLine != null)
7                         break;
8                 }
9             }
10         }
11         return this;
12    }
```

図 12: Nopol によるオーバーフィッティングの例 2

的にテストケースを通過した．しかし，nullText および newLine に何かしらの文字列を代入するようなテストケースを作成すれば，テストケース通過とはならないため，オーバーフィットといえる．図 12 は commons-text における StrBuilder クラスの replaceAll メソッドを示している．replaceAll メソッドは，フィールド変数に格納されている文字列において，char 型変数 search に一致する文字をすべて char 型変数 replace が示す文字に置換するというメソッドである．一方加工対象メソッドの replaceFirst は，char 型変数 search に最初に一致した文字を replace に置換するメソッドである．生成コードでは，6 行目に if 文が追加されている．図 11 の例で述べたように，StrBuilder のテストケース

において，newLine が常に null であるため，7 行目の文が実行されない．したがって，replaceAll と等価な処理になるが，newLine が null でないようなテストケースを作成すれば仕様を満たさなくなる．よってオーバーフィットである．

表 3: Nopol を用いた場合の実験結果

プロジェクト名	自動生成されたメソッド数	正しく自動生成されたメソッド数
commons-text	2	0
commons-lang	0	0
commons-io	0	0
commons-collections	0	0

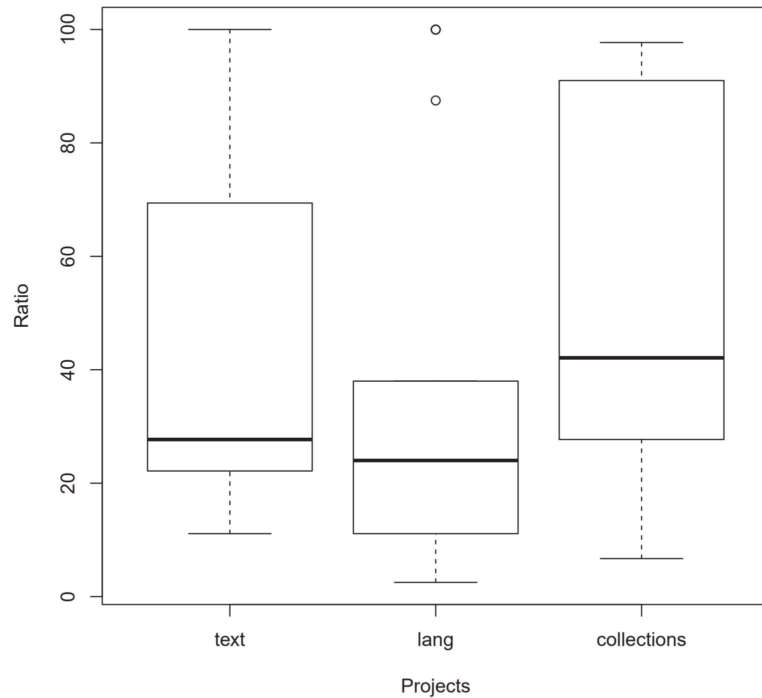

図 13: 自動生成に用いた加工対象メソッドの割合

7　考察

7.1　自動生成されたメソッド

正しく自動生成されたメソッドの多くは，ユーティリティメソッドであった．ユーティリティメソッドは，他クラスのメソッド呼び出しなどの依存関係が比較的少ないシンプルなメソッドである．このことから，依存関係が少ないクラスに存在しているメソッドを加工対象とすると精度の改善につながると考えられる．また，正しく自動生成されたメソッドの中には，無意味なコードが存在しているものがある．可読性や実行効率という観点からみれば，開発者にとっては望ましくないコードとなる．したがって，自動生成されたソースコードは最適化する必要がある．

7.2　オーバーフィッティングなメソッド

オーバーフィッティングであったメソッドの中には，あと 1 文加工すれば正しい自動生成になる，というものがあった．あと 1 文加工できなかった要因としては，テストケースが十分ではなかったことである．改善策として，テストケースを漸進的に追加していく方法が考えられる．本実験では既存のテストケースのみ使用したが，オーバーフィットなメソッドが生成されるたびに，テストケースを順次更新していく作業を追加すれば，目的のメソッドを得られる可能性が高くなる．全自動な手法ではなくなるが，開発者がコードを書く作業は軽減することができる．また，別の改善策としては，jGenProg の精度向上が挙げられる．jGenProg では，加工の際に用いるプログラム文をランダムに選

択している．対象のプログラムが大規模である場合，ランダムに選択すると加工に時間がかかる，もしくは加工に有用なプログラム文を取りこぼす恐れがある．そこで，プログラム文の選択に優先順位をつける方法が考えれる．優先順位のつけ方として，コードの類似度を用いる手法が提案されている [39][40].

7.3 jGenProg と Nopol の比較

既存研究において，プログラム修正に関しては Nopol の方が多くの欠陥を修正することができている [24]．しかし本研究において，メソッドの自動生成に関しては jGenProg の方が多くのメソッドを生成することができた．また本実験において，Nopol が行った加工処理は，文の削除に相当するものであり，jGenProg にも可能な処理である．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が有用であるといえる．

7.4 加工対象メソッドのランク付け

自動生成された各メソッドが，ランク付けさ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 13 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，5.3 節における処理 2 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．

7.5 実行時間

次に，自動生成に要した時間について考察する．自動生成されなかったメソッドは，30 分のタイムアウトによるものか，全ての加工のパターンを試したが最終的にテストケースを通過しなかったことが要因である．それらを除き，自動生成されたメソッドに要した時間の結果を図 14 に示す．全てのメソッドが，タイムアウト時間よりも大幅に短い時間で生成されていることがわかる．このことから，自動生成の精度を改善するためには，タイムアウト時間を延ばすのではなく，一世代あたりの個体数など，遺伝的アルゴリズムの設定を変更する必要があると考えらえる．

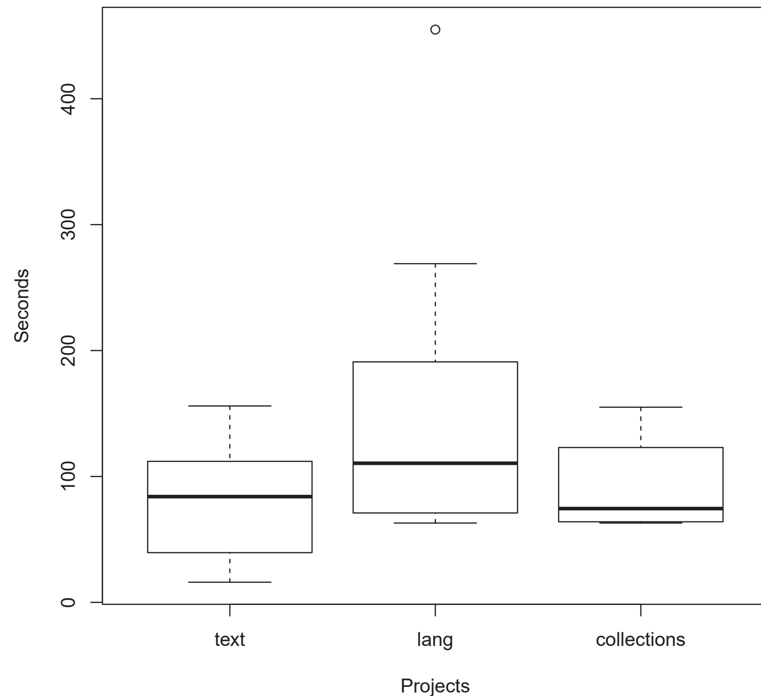

図 14: 自動生成に要した時間

8 妥当性の脅威

8.1 実験対象

本研究では提案手法を Java で開発された 4 つのオープンソースソフトウェアのメソッドに対して適用した．他のオープンソースソフトウェアを実験対象とした場合，異なる結果が得られる可能性がある．

8.2 自動プログラム修正ツール

本研究ではメソッドの加工処理を jGenProg と Nopol を用いて行った．他の自動プログラム修正ツールを用いた場合，異なる結果が得られる可能性がある．

8.3 自動生成されたメソッド

適用実験において生成されたメソッドがオーバーフィットかどうかの判断は目視確認によるものである．そのため，正しい自動生成と判断しているものがオーバーフィットである，もしくはオーバーフィットであると判断しているものが正しい自動生成である可能性がある．

9　あとがき

本研究では，Java メソッドの仕様からソースコードを自動生成する手法を提案した．提案手法では，メソッドのシグネチャ情報と入出力情報を与えることで，目的のメソッドを得ることができる．

提案手法をツールとして実装し，オープンソースソフトウェアに対して適用した．その結果，処理内容が類似したメソッドを加工することによって，自動生成が可能であることが分かった．

今後の取り組みとして，提案手法の評価実験をさらに行っていきたいと考えている．実験の流れとしては，まずプロジェクトの開発履歴を解析する．そして，あるリビジョン r とリビジョン r+1 間において，追加されたメソッドをテストデータとする．つまり，リビジョン r+1 に追加されたメソッドが，リビジョン r 内のメソッドを加工することで自動生成できるかどうかを評価する．

また，提案手法の改善点として，加工可能 (コンパイル可能) なメソッドの拡充が挙げられる．本手法では，加工可能な加工対象メソッドが見つかるまで探索を行ったが，コンパイルできるように加工対象メソッドのソースコードを変換すると，メソッドの自動生成の可能性が高まる．具体的には，フィールド変数のマッピング，引数の順序変更などが挙げられる．他の改善点としては，メソッドのランク付け手法の改良が挙げられる．本手法ではメソッド名の類似度のみを用いているが，変数名や構文情報などの意味的なメトリクスを用いたランク付けも行う必要があると考えている．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本真二教授に心より感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました，柗本真佑助教に深く感謝申し上げます.

本研究を進めるにあたり，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に心より感謝致します.

最後に，本研究に至るまでに，講義やセミナー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の諸先生方に，この場を借りて心より御礼申し上げます.

参考文献

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

[2] Tihomir Gvero and Viktor Kuncak. Synthesizing java expressions from free-form queries. *Acm Sigplan Notices*, Vol. 50, No. 10, pp. 416–432, 2015.

[3] Sumit Gulwani, William R Harris, and Rishabh Singh. 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] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pp. 13–24, New York, NY, USA, 2010. ACM.

[6] 佐藤, 重幸. Pldi 2015 報告. コンピュータ ソフトウェア, Vol. 33, No. 1, pp. 1_15–1_21, 2016.

[7] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pp. 36–46, New York, NY, USA, 2010. ACM.

[8] Luke S. Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2 - Volume 2*, ACL '09, pp. 976–984, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[9] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 215–224, New York, NY, USA, 2010. ACM.

[10] Susumu Katayama. Systematic search for lambda expressions. *Trends in functional programming*, Vol. 6, pp. 111–126, 2005.

[11] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[12] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, Vol. 4, No. 2, pp. 87–112, 1994.

[13] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, Vol. 185, pp. 825–885, 2009.

[14] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

[15] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pp. 171–177. Springer, 2011.

[16] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pp. 404–415, Piscataway, NJ, USA, 2017. IEEE Press.

[17] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. Codecarboncopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 95–105, New York, NY, USA, 2017. ACM.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[19] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 3–13. IEEE, 2012.

[21] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–317. ACM, 2014.

[22] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Softw. Eng.*, Vol. 43, No. 1, pp. 34–55, January 2017.

[23] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pp. 437–440, New York, NY, USA, 2014. ACM.

[24] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 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.

[25] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Steven P. Reiss. Specifying what to search for. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pp. 41–44, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A key to reuse. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '93, pp. 182–190, New York, NY, USA, 1993. ACM.

[28] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pp. 91–100, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[29] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 463–475, 1994.

[30] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. Codegenie: Using test-cases to search and reuse source code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pp. 525–526, New York, NY, USA, 2007. ACM.

[31] Michael D Ernst, Raimondas Lencevicius, and Jeff H Perkins. Detection of web service substitutability and composability. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, pp. 123–135, 2006.

[32] SQLite. https://www.sqlite.org/.

[33] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, pp. 35–44. IEEE, 2011.

[34] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? an exploratory study on java methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 294–305. ACM, 2014.

[35] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 24–36. ACM, 2015.

[36] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pp. 65–74, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Matias Martinez and Martin Monperrus. Astor: a program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441–444. ACM, 2016.

[38] EclEmma. https://www.eclemma.org/.

[39] Haruki Yokoyama, Yoshiki Higo, Keisuke Hotta, Takafumi Ohta, Kozo Okano, and Shinji Kusumoto. Toward improving ability to repair bugs automatically: A patch candidate location mechanism using code similarity. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pp. 1364–1370, New York, NY, USA, 2016. ACM.

[40] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. *arXiv preprint arXiv:1707.04742*, 2017.