

特別研究報告

題目

既存メソッドの再利用・加工による
ユーティリティメソッドの自動生成

指導教員

楠本 真二 教授

報告者

松本 淳之介

平成 30 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 29 年度 特別研究報告

既存メソッドの再利用・加工による
ユーティリティメソッドの自動生成

松本 淳之介

内容梗概

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、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。仕様は、シグネチャ情報 (引数の型と返値の型, メソッド名) およびそのメソッドに関するテストケースである。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。

主な用語

ユーティリティメソッド, 自動プログラミング, 自動生成

目次

1	まえがき	1
2	準備	3
2.1	ユーティリティメソッド	3
2.2	欠陥箇所の限局	3
2.3	遺伝的プログラミング	4
2.4	GenProg	4
3	提案手法	7
4	実装	9
4.1	前準備	9
4.2	メソッドの自動生成	9
5	適用実験	11
5.1	実験概要	12
5.2	実験対象	12
5.3	実験手順	13
5.4	出力されるメソッドの分類	14
5.5	実験結果	14
6	考察	18
6.1	再利用されただけのメソッドに対する考察	18
6.2	加工されたメソッドに対する考察	19
6.3	オーバーフィットなメソッドに対する考察	19
6.4	全体の出力に対する考察	20
7	妥当性への脅威	21
7.1	出力されたメソッドの目視確認	21
7.2	GenProg の Seed 値	21
8	関連研究	22
8.1	Reiss の研究	22

8.2	Balog らの研究	22
8.3	下仲らの研究	23
9	あとがき	24
	参考文献	26

図目次

1	ユーティリティメソッドの例	3
2	GenProg の動作の流れ	5
3	提案手法の概要	7
4	メソッドをコピーして加工する流れ	11
5	対象のメソッドの書き換え例	13
6	差異が引数の null チェックのみのメソッドの例	14
7	再利用されただけのメソッドの例	15
8	加工されたメソッドの例	17
9	オーバーフィットなメソッドの例	18
10	private なメソッドが呼び出される例	20

表目次

1	出力されたメソッドの内訳	15
---	------------------------	----

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[5] 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[12] 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。このことからユーティリティメソッドが多く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。詳しくは 8 章で述べるが、コードの自動生成に関する研究は様々である。例えば、機械学習を用いたコードの自動生成 [14] があるが、未だに実用的なものではない。本研究で生成の対象となるものはユーティリティメソッドである。ユーティリティメソッドは汎用的な処理をするメソッドなため、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同じ引数の型、戻り値の型 (以降、戻り値の型と引数の型の組み合わせをシグネチャと呼ぶ) を持つ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通りの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。本研究における仕様とは、以下の組み合わせのことを意味する。

- シグネチャ情報
- 自動生成対象のメソッドに関するテストケース

提案手法を評価するために、オープンソースソフトウェアに存在している既存のユーティリティメソッドを、提案手法を用いて自動生成することができるか実験を行った。既存の 176 個のユーティリティメソッドに対して自動生成を試み、16 個のユーティリティメソッドの自動生成に成功した。

以降、2 章では本研究の提案手法に必要な事柄について説明する。3 章ではユーティリティメソッドの自動生成を実現させる手法を提案する。4 章では提案手法の実装について述べる。5 章では提案手法の適用実験について述べ、6 章で実験結果についての考察を行う。7 章で妥当性の脅威について述べ、8

章で関連研究について述べ、9章で本研究のまとめと今後の課題について述べる。

```
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

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し、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[2] 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 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の候補を探す手法であり、最も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 [13]。疑惑値とは、欠陥である可能性の高さを表す値である。このような手法では、通過済みテストのみが実行する文は疑惑値が低くなり、未通過テストのみが実行する文は疑惑値が高くなる。また、GenProg およびその関連手法のうちいくつかはプログラムの変更を行う際、欠陥箇所の限局を行い、疑惑地の高いプログラム文を優先的に変更する。

2.3 遺伝的プログラミング

遺伝的プログラミングとは自然界の生物が環境に適応して進化していく過程を模した探索アルゴリズムである [19]。データ（解の候補）を生物の個体に見立て、生物の進化の過程で行われる事象をモデル化した遺伝的操作と呼ばれるいくつかの操作を繰り返し適用することで、環境に適した、できるだけ最適解に近い個体を生成することを目的とする。

遺伝的プログラミングでは、まず一定数の個体をランダムに生成し、これらを第 1 世代とする。これらの個体に対して遺伝的操作を適用することで次世代の個体群を生成する。同様にして新たな世代の個体群の生成を繰り返す。このような処理を、条件を満たす個体が生成されるか、あらかじめ定められた世代数に到達するまで繰り返す。

遺伝的プログラミングで一般的に行われる遺伝的操作は以下の 3 つである。

選択 生物の自然淘汰をモデル化したもので、環境への適応度に基づいて一定数の個体を取り出す。環境への適応度は解への近さを表す値で、評価関数に基づき算出される。評価関数は実装により異なる。

変異 生物に見られる遺伝子の突然変異をモデル化したもので、個体に対して何かしらの変更を加える。

交叉 生物が交配によって子孫を残すことをモデル化したもので、2 つの個体を混ぜ合わせた新たな個体を生成する。

2.4 GenProg

GenProg は遺伝的プログラミング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 [20]。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

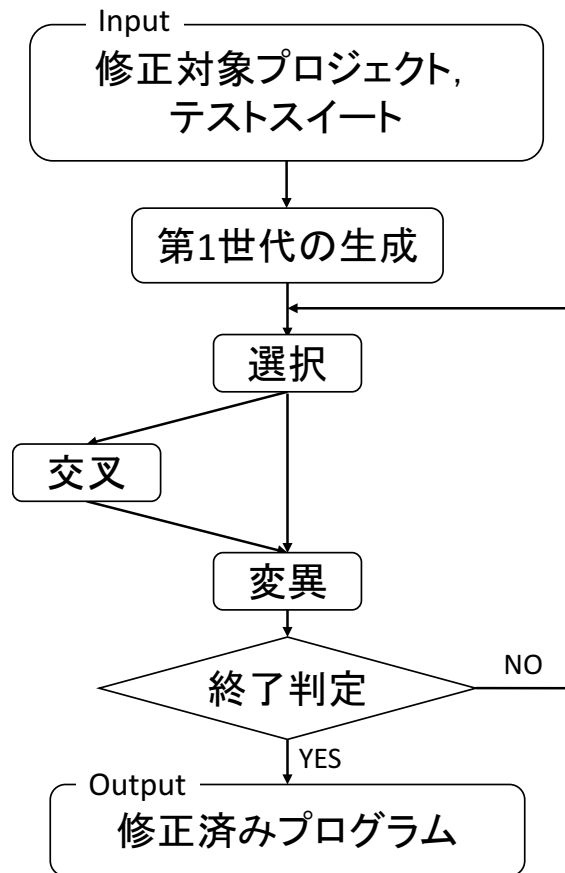


図 2: GenProg の動作の流れ

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の変異プログラム群と呼ぶ。

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 [15]。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。

GenProg は元々 OCaml で開発されているが、Java で再実装された jGenProg[3] 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[22], MutRepair[17]) を Java で再実装し、Astor というツールにまとめている [21]。本研究では Astor に内包されている jGenProg を用いる。

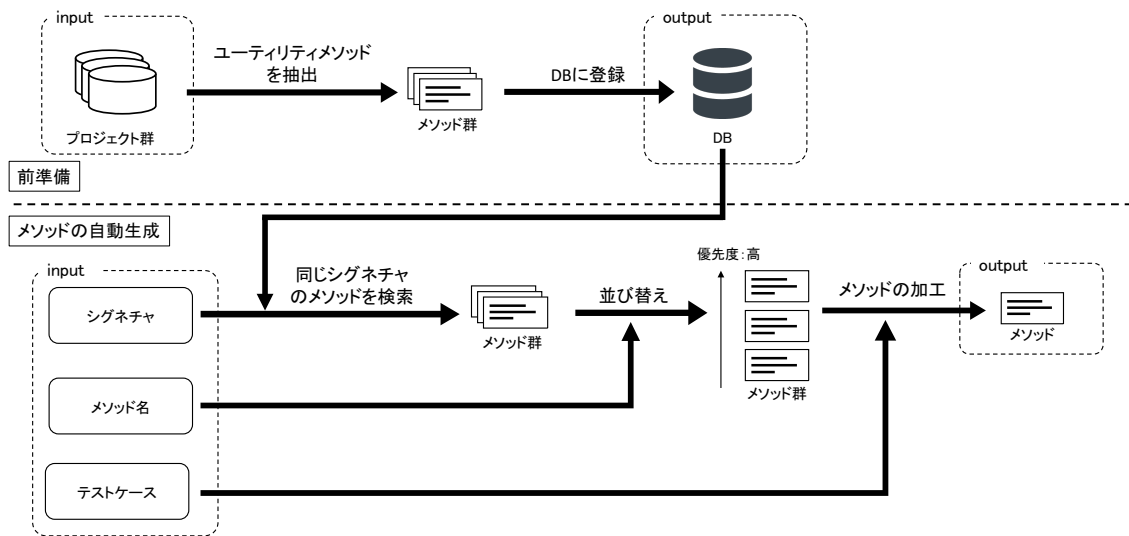


図 3: 提案手法の概要

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シングネチャ情報、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ユーティリティメソッドを抽出し、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ン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。本研究の実装では、DB として SQLite[25] 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
- 戻り値の型
- 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 [16][18] 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。仕様を満たすメソッドを生成できない場合は、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。

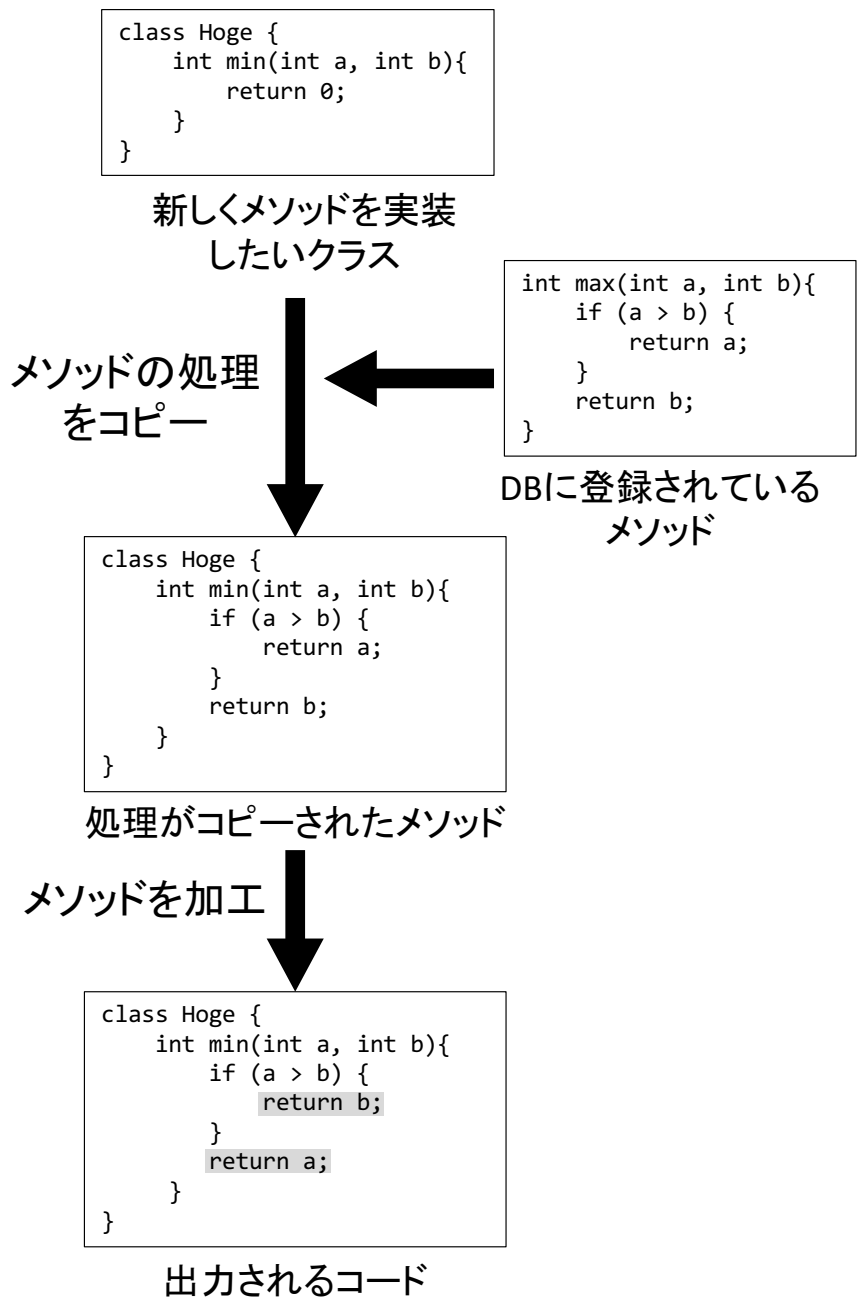


図 4: メソッドをコピーして加工する流れ

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することができるかどうか実験した。

5.2 実験対象

5.2.1 メソッドを格納する DB の構築

今回は実験をするにあたり GitHub[12] に公開されている Java プロジェクトを対象とし、スター上位の Java プロジェクト 30 個に加えて、以下の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。

- apache/commons
- math
- util
- algorithm
- competitive

これらの Java プロジェクトを入力とし、提案手法の前準備を行った。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[10] 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のメソッドとした。

- 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
- 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ことを確かめる実験であるため、自動生成対象のメソッド

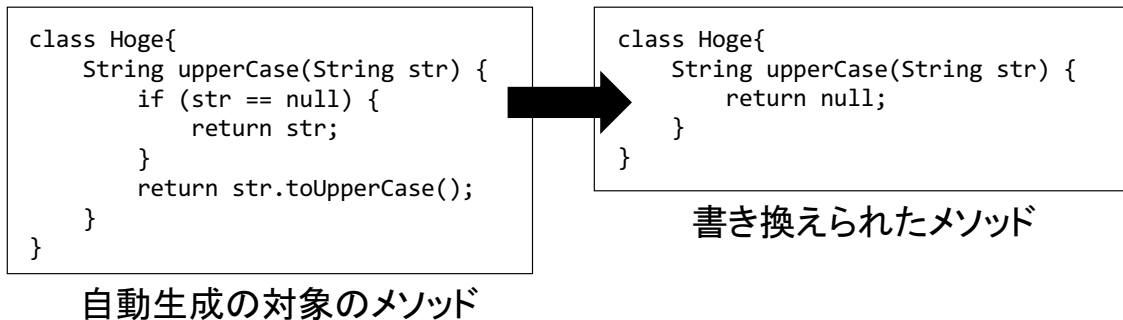


図 5: 対象のメソッドの書き換え例

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次のような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. 対象のメソッドの処理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

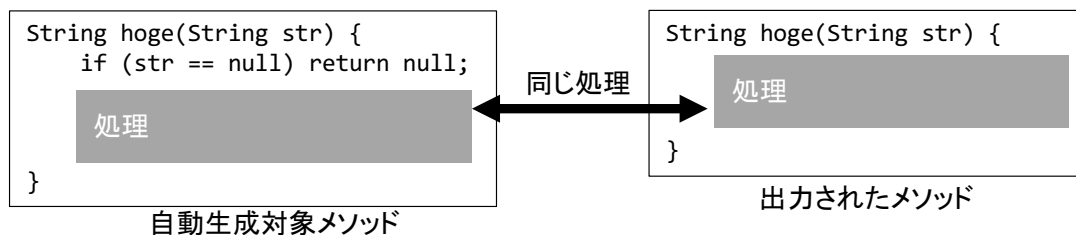


図 6: 差異が引数の null チェックのみのメソッドの例

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しない状態で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： 再利用されるメソッドを加工しない状態ではテストケースを通過せず、自動プログラム修正によって加工され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

今回の実験における自動生成に成功したメソッドとは再利用されただけのメソッド、加工されたメソッドの場合である。オーバーフィットなメソッドが出力された場合は自動生成に失敗したものとして扱う。

ただし図6のように、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

```

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

```

(a) 自動生成対象のメソッドのソースコード

```

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のソースコード

図 7: 再利用されただけのメソッドの例

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[8] 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 7 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[9] 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。この例では加工を加え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、出力されたメソッドを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[7] 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

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

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 8 に示す。このメソッドを生成するにあたり、fastjson[1] 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 8 の (b) と比べると、図 8 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。この例の場合、再利用されたメソッドに加工が加えられているので、出力されたメソッドを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

自動生成対象のメソッドは commons-jxpath[6] 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と再利用されたメソッドのソースコードを図 9 に示す。このメソッドを生成するにあたり、dubbo[4] というプロジェクトの isEqual メソッドが再利用された。しかし、図 9 の (a) と図 9 の (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いるため、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。

```

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のソースコード

```

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のソースコード

```

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のソースコード

図 8: 加工されたメソッドの例

```

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のソースコード

```

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のソースコード

```

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

```

(c) 再利用されたメソッドのソースコード

図 9: オーバーフィットなメソッドの例

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ユーティリティメソッドが生成されていることが重要であり、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ないため、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 の精度が十分になかった

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 の性能の問題がある。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 [15]。そのため、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 [20]。

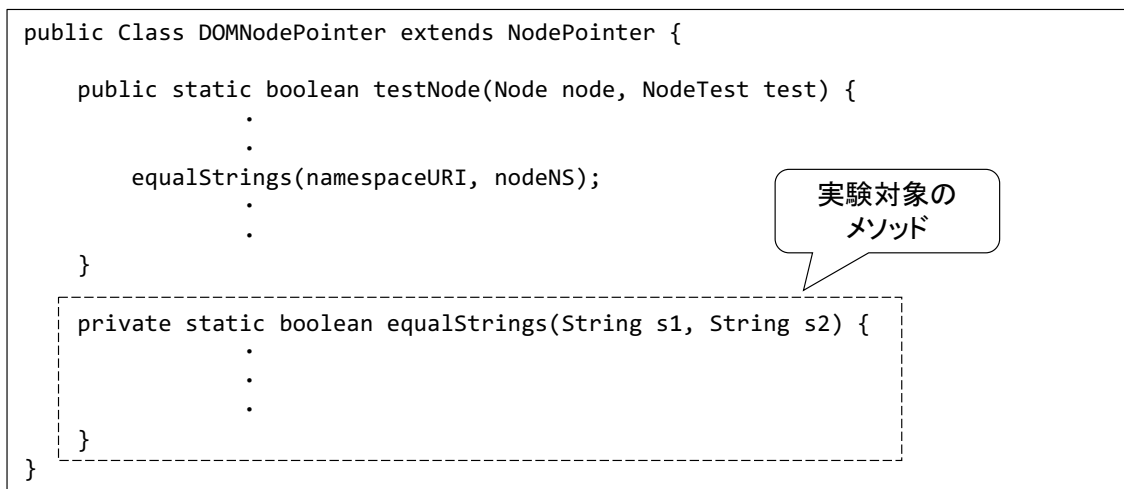
今後の GenProg の発展に応じて、加工されたメソッドの出力数も増えることが考えられる。

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いえる。

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。5.5 章のオーバーフィットなメソッドの例として示した `equalStrings` メソッドが定義されているクラスを図 10 に示す。`private` なメソッドは直接的にはテストの対象にならない。そのため `private` なメソッドは図 10 のように、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。今回の実験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。

```
public Class DOMNodePointer extends NodePointer {  
    public static boolean testNode(Node node, NodeTest test) {  
        .  
        .  
        equalStrings(namespaceURI, nodeNS);  
        .  
        .  
    }  
    private static boolean equalStrings(String s1, String s2) {  
        .  
        .  
        .  
    }  
}
```

A diagram showing a code block with a dashed box around a private method and a callout bubble pointing to it. The callout bubble contains the text "実験対象のメソッド" (Target Method).

実験対象のメソッド

図 10: private なメソッドが呼び出される例

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 妥当性への脅威

7.1 出力されたメソッドの目視確認

本研究ではメソッドの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。

7.2 GenProg の Seed 値

今回はメソッドの加工に GenProg を用いた。GenProg の実行時に Seed 値の設定が必要である。GenProg はこの Seed 値を元に乱数を生成し，プログラムの加工を行う。本研究での実験では 1~4 の計 4 種類の Seed 値で実験を行った。この Seed 値が変われば実験結果が異なる可能性がある。

8 関連研究

プログラムの自動生成に関する研究は様々なものがある。本章ではプログラムの自動生成に関する研究について説明し、本研究との差異について述べる。

8.1 Reiss の研究

Reiss は Web 上に存在するソースコードを収集し、仕様を満たすようメソッドを加工する手法を提案し [24], S^6 というツールを開発した [23]。このツールでは、後方互換があるようなシグネチャの変換や、既存のメソッドから一部のコードを抽出して新しいメソッドを作成するなどといった加工が施される。この加工は、開発者が Web 上に存在している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、ツールに模倣させたものである。そのためメソッドの内部の処理を書き換えて別の処理をするメソッドを新たに生成するといったことは行っておらず、その点が本研究との差異といえる。

8.2 Balog らの研究

Balog らは機械学習を用いたプログラムの自動生成の手法を提案し、DeepCorder というツールを開発した [14]。この手法には次の工程がある。

1. 既存のプログラムを収集し、学習データを作成する
2. 1. で作成した学習データを用いて自動生成したいプログラムに近いプログラムを用意する
3. 複数の探索アルゴリズムを用いて、2. で用意したメソッドが仕様を満たすよう加工していく

プログラムが学習を行うために、DeepCorder では既存のプログラムを学習しやすい別の言語に置き換える必要がある。また、出力されるプログラムもその言語で記述されたプログラムが出力される。その言語にはプログラムに学習させるために様々な制限がある。例えば、DeepCoder のために設計された言語には for 文などといったループが存在せず、ループを用いたプログラムを出力をすることができない。DeepCorder にはこのような制限があり、未だ実用性にかけてプログラムしか出力することができない。

本研究では、Java で書かれたメソッドをそのまま再利用・加工するため、ループの処理をするメソッドを出力することができる。実際、5.5 章で示した実験結果のメソッドの中には、図 7 のようにループを用いたメソッドが出力されている。

8.3 下仲らの研究

下仲らは既存のメソッドを再利用し，GenProg を用いてメソッドを自動生成する手法を提案した [27]．この手法は本研究の提案手法と非常によく似た手法である．しかし，下仲らの研究で対象にしているメソッドは自動生成したいメソッドと同一のプロジェクトに定義されているメソッドに限定している．そのため，自動生成したいメソッドと同一のシグネチャを持つ上，自動生成したいメソッドに近い処理をするメソッドが，自動生成したいメソッドと同一のプロジェクトに定義されていなければメソッドを自動生成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．よって，本研究の提案手法の方がより多くのメソッドを再利用の対象として自動生成することができる．

9 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。

シグネチャの異なるメソッドの再利用 本研究ではシグネチャが全く同じメソッドのみを再利用している。8 章で説明した S^6 のように後方互換性のあるシグネチャも再利用対象にすることで、より多くのメソッドを再利用の対象にすることができる。例えば、int を引数にもつメソッドを生成するときには、Integer を引数にもつメソッドも再利用の対象にすることができる。他には、Object を引数にもつメソッドに対して、Object を継承した String を再利用の対象にすると言ったことである。

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より自動生成に成功することが予想される。

自動プログラム修正のツール： 本研究では自動プログラム修正の技術として GenProg を用いたが、GenProg の他にも Xuan らが提案した Nopol という自動プログラム修正のツールが存在する [26]。Nopol は if 文の追加、および if 文の条件式の変更をする自動プログラム修正ツールである。Martinez らによって GenProg と Nopol の比較実験がされており、Nopol の方が多くの欠陥を修正でき、自動プログラム修正の技術としては Nopol の方が優れていることがわかっている。その Nopol を用いることで、自動生成できるメソッドの数が増え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，松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，適切なご助言および多大なるご助力を頂きました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博士前期課程 2 年の下仲健斗氏に心より感謝申し上げます。

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。

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。

最後に 23 年にわたり育ててくれ，様々な費用を出していただき，大学生活を支えてくれた両親や祖父母にも心より感謝申し上げます。

参考文献

- [1] GitHub - A fast JSON parser/generator for Java.
- [2] Github - a memory leak detection library for android and java.
- [3] Github - automatic program repair for java with generate-and-validate techniques - jgenprog (genprog for java) - jkali - jmutrepair - deeprepair - cardumen.
- [4] Github - dubbo is a high-performance, java based, open source rpc framework.
- [5] Github - google core libraries for java.
- [6] Github - mirror of apache commons jxpath.
- [7] GitHub - Mirror of Apache Commons Lang.
- [8] GitHub - Solutions for some common algorithm problems written in Java.
- [9] GitHub - ZXing (Zebra Crossing) barcode scanning library for Java, Android.
- [10] JaCoCo - Java Code Coverage Library.
- [11] Maven - welcome to apache maven.
- [12] The world's leading software development platform · GitHub.
- [13] Rui Abreu, Peter Zoetewey, Rob Golsteijn, and Arjan J. C. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.
- [14] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs, 2016.
- [15] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 306–317, 2014.
- [16] 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, CSMR '11*, pp. 35–44, Washington, DC, USA, 2011. IEEE Computer Society.
- [17] 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.
- [18] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from pro-

- gram source code? an exploratory study on java methods. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pp. 294–305, New York, NY, USA, 2014. ACM.
- [19] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, Vol. 1. MIT press, 1992.
- [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 *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pp. 3–13, Piscataway, NJ, USA, 2012. IEEE Press.
- [21] 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.
- [22] 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.
- [23] Steven Reiss. Integrating s6 code search and code bubbles, 05 2013.
- [24] 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.
- [25] SQLite. <https://www.sqlite.org/>.
- [26] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34–55, Jan 2017.
- [27] 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, 1 2018.