特別研究報告

題目

プログラム構造が自動生成テストの網羅率に与える影響の調査

指導教員

楠本 真二 教授

報告者

渡邉 凌雅

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

プログラム構造が自動生成テストの網羅率に与える影響の調査

渡邉 凌雅

内容梗概

ソフトウェア開発工程の 1 つに単体テストがある．この工程では，関数やメソッドといったプログラムを構成する単位（ユニット）が開発者の想定通りに振る舞うかどうかを検証する．プログラム開発後の早期の段階でバグや問題を特定できるため，単体テストはソフトウェア開発において欠かせない工程となっている．単体テストを実施するためには，テスト対象ユニットに対応するテストスイートを用意する必要がある．しかし，ユニット全体を網羅できるテストスイートを作成する作業には多大な労力を要する．そのため，単体テストのテストスイートを自動的に生成する研究が行われており，その成果として，さまざまなテスト生成ツールが公開されている．しかし，これらのツールは，入力したプログラムに対しそのプログラムを完全に網羅したテストスイートを生成できない場合がある．そこで本研究では，テスト生成ツールが生成したテストスイート（以下，生成テストスイート）で網羅できない原因の解明を目的として，入力プログラムの構造が生成テストスイートの網羅率に与える影響を調査する．調査の結果，生成テストスイートで網羅できない原因として 4 つのパターンを発見した．また，一部のパターンに対し，網羅できない原因を解決するリファクタリング手法を考案した．

主な用語

単体テスト，テスト生成ツール，網羅率，リファクタリング

目次

図目次

表目次

1　はじめに

単体テストとは，ソフトウェア開発工程の 1 つで，関数やメソッドといったプログラムを構成する単位（ユニット）が開発者の想定通りに振る舞うかどうかを検証する工程である．プログラム開発後の早期の段階でバグや問題を特定できるため，単体テストはソフトウェア開発において欠かせない工程となっている．単体テストを実施するためには，テスト対象ユニットに対応するテストスイートを用意する必要がある．しかし，ユニット全体を網羅したテストスイートを作成する作業は多大な労力を要することから，単体テストのテストスイートを自動的に生成する研究が行われている [1]．その成果として，EvoSuite [2] や Randoop [3]，SUSHI [4] といったテスト生成ツールが公開されている．

しかし，既存のテスト生成ツールでは，テスト対象ユニットを完全に網羅したテストスイートを生成できない場合がある．この問題は，テスト生成ツールの生成アルゴリズムやテスト生成時のパラメータ設定，あるいは入力プログラムの構造など，さまざまな要因に左右されると考えられる．本研究では，それらの要因の 1 つとして入力プログラムの構造に着目する．

本研究では，テスト生成ツールで生成されたテストスイート（以下，「生成テストスイート」と呼ぶ）で網羅できないコードを持つメソッド 73 件について，2 つの Research Question（RQ）を設定し調査を実施する．

RQ1: 生成テストスイートがコードを網羅できない原因は何か？

生成テストスイートで網羅できないコードを持つメソッドに対し，そのコードが網羅されなかった原因を調査する．その後，そのコードを網羅できない原因の共通点を考察し，得られた共通点をもとにパターン化する．調査の結果，網羅できない原因に見られる特徴として，「特定の値を要求」「特定の型を要求」「実行不可能コード」「マルチスレッド処理」の 4 つのパターンに分類できることを発見した．

RQ2: 網羅できない原因を解決するリファクタリング手法は存在するか？

RQ1 の調査で得られた原因パターンに対し，網羅できない原因を解決するようにコードをリファクタリングする手法を提案する．本研究では，「特定の値を要求」に該当するメソッドの一部に適用可能なリファクタリング手法と，「実行不可能コード」に該当する全メソッドに適用可能なリファクタリング手法を考案した．

2 準備

2.1 単体テスト

単体テストとは，ソフトウェア開発工程の 1 つで，関数やメソッドといったプログラムを構成する単位（ユニット）が開発者の想定通りに振る舞うかどうかを検証する工程である．プログラム開発後の早期の段階でバグや問題を特定できるため，単体テストはソフトウェア開発において欠かせない工程となっている．

単体テストでは，作業効率化のためにテスト自動生成フレームワークが使われる．テスト自動化フレームワークとは，単体テストを実行するためのソフトウェアおよびテストケースの記述方法を提供するフレームワークであり，作成したテストを実行する工程の自動化や，テスト実行結果レポートの自動生成などが実行できる．Java 用のテスト自動化フレームワークとしては JUnit[*1]があり，Eclipse[*2]や Intellij IDEA[*3]といった代表的な Java 向け統合開発環境でサポートされている．

単体テストを実施するためには，テスト対象ユニットに対応するテストスイートを用意する必要がある．テストスイートとは，何らかのテスト目標を達成するために作成されたテストケースの集合を指す．テストケースとは，テスト項目の最小単位であり，テスト対象への入力と想定される結果の組み合わせで構成される．

JUnit におけるテストスイートの例を図 1 を用いて説明する．図 1a は，この例においてテスト対象となるクラス `Sample` である．このクラスに含まれるメソッド `isOdd()` は，入力された値が奇数であるかどうかを返すメソッドである．図 1b は，`Sample` クラスをテストするためのクラス `SampleTest` である．`SampleTest` に含まれるメソッド `testEven()` は，`Sample.isOdd()` に対し正の偶数を入力したときの動作を検証するテストケースである．この場合，想定結果は `false` となるため，`assertFalse()` で返り値が `false` であると主張する．

[*1] `https://junit.org/junit5/`

[*2] `https://www.eclipse.org/`

[*3] `https://www.jetbrains.com/ja-jp/idea/`

```java
public class Sample {
    public boolean isOdd(int n) {
        boolean result = false;
        if (n % 2 != 0) {
            result = true;
        }
        return result;
    }
}
```

(a) テスト対象クラスの例

```java
import org.junit.Test;
import static org.junit.Assert.assertFalse;
public class SampleTest {
    @Test
    public void testEven() throws Exception {
        Sample sample = new sample();
        int n = 2;
        boolean result = sample.isOdd(n);
        assertFalse(result);
    }
}
```

(b) テストスイートの例

図 1: テスト対象クラスとテストスイートの例

2.2 網羅率

網羅率とは，テスト対象ユニットにおいてテストされた割合を表す指標であり，単体テストにおけるテストケースの品質を測る尺度として用いられる．網羅率を計測する際の基準を網羅基準と呼ぶ．網羅基準にはさまざまな種類があり，代表的な基準として命令網羅（C0），分岐網羅（C1），条件網羅（C2）がある．2.4 節で説明するテスト生成ツールでは，命令網羅の代わりに行網羅が用いられ，また分岐網羅も用いられる．そのため，ここでは行網羅と分岐網羅について説明する．

行網羅

行網羅では，テスト対象ユニットに含まれる実行可能な行のうち，テストスイートによって実行された割合を計測する．命令網羅によって網羅率を計測する場合，

$$\text{網羅率} = \frac{\text{テストスイートによって実行された行数}}{\text{テスト対象ユニットに含まれる実行可能な行の総数}}$$

という式で計測する．

分岐網羅

分岐網羅では，テスト対象ユニットに含まれる分岐に対し，その分岐条件が真となる場合と偽となる場合をテストスイートによって網羅できたかどうかで計測する．分岐網羅によって網羅率を計測する場合，

$$\text{網羅率} = \frac{\text{テストスイートによって実行された分岐方向の数}}{\text{テスト対象ユニットに含まれる分岐方向の総数}}$$

という式で計測する．

これらの 2 つの網羅基準を比較すると，行網羅よりも分岐網羅の方が網羅するための条件が厳しい．例として，図 1a に含まれるメソッド `isOdd()` を完全に網羅するために必要なテストケースを説明する．表 1 は，2 つのテストスイートを考えた際に，それぞれのテストスイートによって各網羅基準を完全に網羅できるかどうかを示している．テストスイート 1 では，5 行目の分岐が真となり 6 行目を網羅できるため，行網羅を完全に網羅できる．一方で，5 行目の `if` 文の `false` 分岐を網羅していないため，分岐網羅を完全に網羅できない．そのため，分岐網羅を網羅するためには，テストスイート 2 のように奇数を入力するテストケースの他に偶数を入力するテストケースが必要である．偶数の入力によって 5 行目の分岐の両方の分岐方向を網羅できるため，行網羅と分岐網羅を完全に網羅できる．

2.3 網羅率計測ツール

網羅率計測ツールとは，単体テストにおけるテストスイートの網羅率および網羅箇所を計測し可視化するツールである．Java 言語向けのツールとしては JaCoCo[*4]や Clover[*5]がある．

網羅率計測ツールが出力するレポートの例を示す．図 2 は，図 1a の `Sample` クラスに対し図 1a のテストスイートを実行し，その結果を JaCoCo で出力した際に得られるカバレッジレポートである．命令や分岐がテストスイートによって網羅された場合は緑で，分岐方向の一部だけが網羅された場合は黄

表 1: 網羅基準と網羅に必要なテストスイートの関係

テストスイート		`isOdd()` を完全に網羅できるか	
番号	入力	行網羅	分岐網羅
1	奇数	○	×
2	奇数 偶数	○	○

[*4] `https://www.jacoco.org/jacoco/`
[*5] `https://openclover.org/`

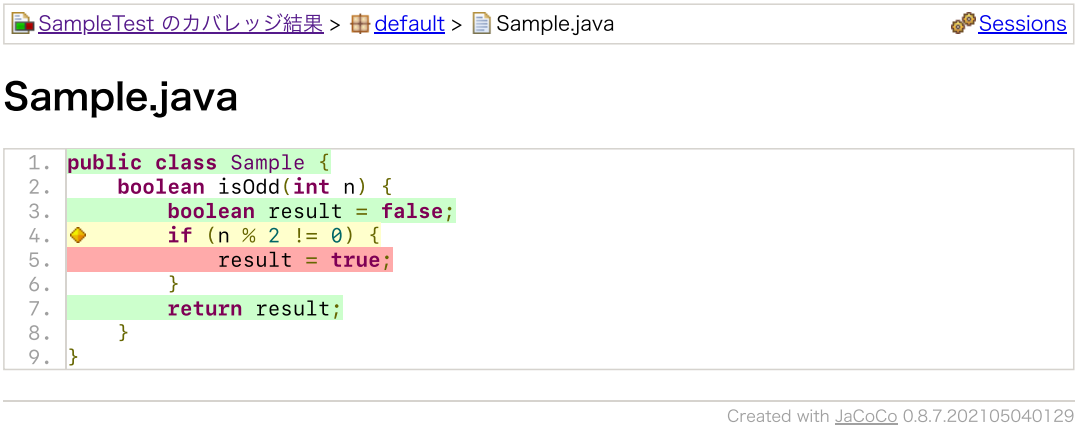

図 2: JaCoCo のカバレッジレポート出力例

で，命令や分岐を完全に網羅できなかった場合は赤でハイライトされる．このように，テストスイートによる網羅の可否が行単位で可視化されるため，網羅率計測ツールは，網羅できなかったコードを特定したり，どのようなテストケースが不足しているかを確認したりするのに役立つ．

2.4 テスト生成ツール

一般に，単体テストのテストスイートを作成する作業は多大な労力を要することから，単体テストのテストスイートを自動的に生成する研究が行われている [1]．その成果として，EvoSuite [2] や Randoop [3]，SUSHI [4] といったテスト生成ツールが実際に利用可能な状態で公開されている．テスト生成ツールがテストスイートを生成する方針はツールによってさまざまである．例えば，EvoSuite では遺伝的アルゴリズムを使用する一方，Randoop ではランダムテストを使用する．また，SUSHI では遺伝的アルゴリズムに加えてシンボリック実行を使用する．

本研究で使用した EvoSuite について説明する．EvoSuite は，遺伝的アルゴリズムをベースに，ハイブリッド検索 [5] や動的シンボリック実行 [6]，テスト容易性変換 [7] といった技術を利用することで，網羅率の高い JUnit テストスイートを生成するツールである．EvoSuite は，さまざまな Java 向けのテスト生成ツールの中でも特に網羅率の高いテストスイートを生成できる点が特徴[*6]である．

EvoSuite がテストスイートを生成する手順について説明する．EvoSuite がテストスイートを生成する際の動作の概略を図 3 に示す．EvoSuite で使われる遺伝的アルゴリズムでは，テストスイートに含まれるテストケースが進化時の個体に対応する．まず，入力したプログラムをもとにランダムなテスト

[*6] EvoSuite は，2021 年に開催されたテスト生成ツールのコンペティションにおいて最も高い総合スコアを獲得している [8]．

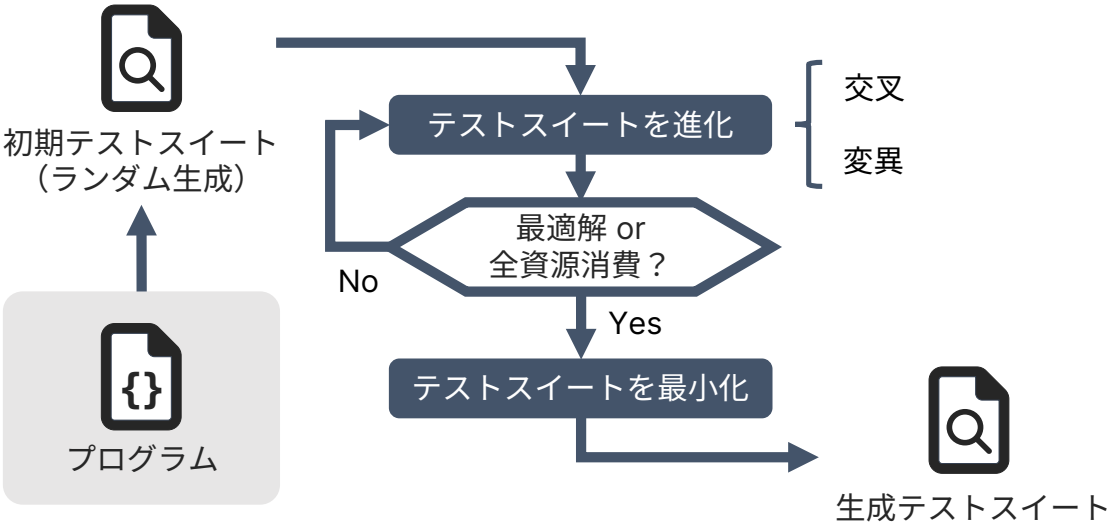

図 3: テストスイート生成における EvoSuite の動作

ケースを生成し，それらの集合を初期テストスイートとする．次に，初期テストスイートを初期世代として，ある世代が最適解に達するか，割り当てられた計算資源を使い果たすまでテストスイートを進化させ続ける．進化過程では，親世代の個体に対し交叉や変異を施した個体が子孫の個体となる．交叉は，子孫の個体の誕生時に命令の一部を入れ替える操作を指す．変異は，子孫の個体の誕生時に既存の命令を変更あるいは削除するか，新たな命令を追加する操作を指す．最後に，進化過程によって得られた最終世代のテストスイートの各テストケースに対し不要な命令を削除してテストスイートを最小化する．

EvoSuite では，行網羅や分岐網羅，例外網羅といったさまざまな網羅基準を最大限満たすようにテストスイートを生成する [9]．ここで EvoSuite は，代表的な網羅基準である命令網羅ではなく，網羅できた行を基準とする行網羅と呼ばれる網羅基準を使用する．EvoSuite はテスト生成時にバイトコード表現を参照する．しかし，バイトコード命令と実際のソースコード上の命令が直接対応しない場合があるため，命令網羅では網羅対象を正確に計測できない．そのため，EvoSuite に限らずテスト生成時にバイトコード表現を参照するツールでは，命令網羅の代わりに行網羅を使用するのがデファクトスタンダードとなっている [2].

3 Motivating Example

テスト生成ツールは，入力したプログラムに対し，そのプログラムを完全に網羅したテストスイートを生成できない場合がある．この問題は，テスト生成ツールの生成アルゴリズムやテスト生成時のパラメータ設定，あるいは入力プログラムの構造など，さまざまな要因に左右される．本研究では，それらの要因の 1 つとして入力プログラムの構造に着目する．

この問題を示す例として，EvoSuite が生成したテストスイートで網羅されないコードを含むメソッドの例を図 4[*7]に示す．図 4 に含まれるハイライトは，3 行目の条件文 `val == null` の `false` 分岐および 5 行目以降の命令を実行できなかったことを表している．

テスト生成ツールが生成したテストスイートによってこれらのコードが実行されなかった原因について考察する．5 行目以降が実行されない理由は，3 行目の条件文 `val == null` の `false` 分岐が実行できていないためである．そこで，この条件文に登場する変数 `val` に着目すると，この変数は 2 行目の `System.getProperty(prop)` の返り値である．つまり，3 行目の条件が実行されない理由は，`System.getProperty(prop)` が `null` 以外の値を返さないためである．`System.getProperty()` メソッドは，実際のシステムプロパティ名（`"user.dir"`など）を入力するとそのプロパティの値を返し，それ以外の文字列を入力すると `null` を返すという仕様である．ここで，このメソッドから EvoSuite が生成したテストケースの例を表 2 に示す．表 2 を参照すると，実際のシステムプロパティ名を持つテストケースが存在しないことがわかる．

以上より，テスト生成ツールが生成したテストスイートによって実行できなかった原因は，`System.getProperty()` メソッドを使用したことで，EvoSuite が入力に実際のシステムプロパティ名を持つテストケースを生成できなかったためと結論づけられる．

[*7] 本稿では，生成テストスイートによって一部だけ実行されたコードを 黄色 で，実行されなかったコードを 橙色 で表す．

```
1  boolean getBooleanProperty(String prop, boolean defaultVal) {
2      String val = System.getProperty(prop);
3      if ( val == null )
4          return defaultVal;
5      if ( val.equalsIgnoreCase("true") ) {
6          return true ;
7      } else {
8          return false ;
9      }
10 }
```

図 4: テスト生成ツールが生成したテストスイートで実行されないコードを持つメソッドの例

表 2: 図 4 のメソッドから生成されるテストケースの例

入力	アサーション
("7G", false)	false
(null, false)	NullPointerException
("", false)	IllegalArgumentException
("Hwiz5]f", true)	true

4 Research Questions

本研究では，プログラム構造が自動生成テストの網羅率に与える影響を調査する．調査にあたって，次の 2 つの Research Question（RQ）を設定した．

RQ1: 生成テストスイートがコードを網羅できない原因は何か？
RQ2: 網羅できない原因を解決するリファクタリング手法は存在するか？

RQ1: 生成テストスイートがコードを網羅できない原因は何か？

生成テストスイートで網羅できないコードを持つメソッドに対し，そのコードが網羅されなかった原因を調査する．その後，網羅できない原因の共通点を考察しパターン化する．

RQ2: 網羅できない原因を解決するリファクタリング手法は存在するか？

RQ1 の調査で得られた原因パターンに対し，網羅できない原因を解決するようにコードをリファクタリングする手法を提案する．

5　調査準備

調査にあたって実施した準備について説明する．まず，複数の Java メソッドで構成されるデータセット [10] に対し EvoSuite でテストスイートを生成する．次に，調査対象となる生成テストスイートで網羅できないコードを持つメソッドを抽出する．

5.1　手順

調査準備の手順を以下に示す．

1. データセットに含まれる各メソッドに対し，そのメソッド単体で構成されるクラスを作成する．このとき，すべてのクラスをコンパイルできるように，クラスファイルの先頭に `import java.lang.*;` と `import java.util.*;` を挿入する．
2. 各クラスに対し EvoSuite でテストスイートを生成する
3. JaCoCo を用いてテストスイートの網羅率を計測する
4. 行網羅率と分岐網羅率の少なくとも一方が 1.0 を下回るメソッドを抽出する

5.2　データセット

本研究では，調査対象として Java 言語で記述されたメソッド 768 件で構成されるデータセット [10] を使用する．このデータセットに含まれるメソッドはすべて以下のような特徴を持つ．

- メソッド内部で処理が完結する．すなわち，メソッドの処理がメソッドの外にある変数やメソッドに依存しない．
- 1 つ以上の引数を持ち，処理終了時に何らかの値を返す．すなわち，引数を持たないメソッドや返り値を持たないメソッドは含まれない．
- Java8 までの言語仕様で記述される．
- Java の標準パッケージである `java.lang` および `java.util` 以外を使用しない．

5.3　網羅基準

2.4 節で説明したように，EvoSuite では，テストスイート生成時の網羅目標としてさまざまな網羅基準を設定している．そのため，どの網羅基準に着目するかによって調査結果が大きく異なると考えられる．そこで，本研究では，以下の理由により行網羅と分岐網羅を網羅基準に着目する．

- 実世界における代表的な網羅基準であり，調査する意義が大きいと考えたため

- JaCoCo による網羅率の計測と網羅箇所の可視化に対応しているため

5.4 結果

調査準備の結果，データセットに含まれるメソッド 768 件のうち，約 9.5 %に相当する 73 件のメソッドを抽出した．なお，成功テストを生成できなかったメソッドや，JaCoCo のバグにより網羅されていないと誤判定されたメソッドは除外した．

6　RQ1 に対する調査

生成テストスイートで網羅できないコードを持つメソッドに対し，そのコードが網羅されなかった原因を調査した．その後，網羅できない原因の共通点を考察しパターン化した．

6.1　調査方法

5 節の手順で抽出したメソッドに対し，プログラムの構造や網羅できないコードの特徴を目視で確認し，生成テストスイートがコードを網羅できなかった原因を考察した．そして，各メソッドを調査して得られた結果をもとに，網羅できない原因に見られる共通点を考察しパターン化した．

6.2　結果

調査の結果として，生成テストスイートが何らかのコードを網羅できなかった原因に見られるパターンを表 3 に示す．そのようなコードを含むメソッド 73 件のうち，全体の約 79% に相当する 58 件が「特定の値に要求」のパターンに分類された．残りの 15 件のメソッドは，「実行不可能コード」「特定の型を要求」「マルチスレッド処理」の 3 パターンに分類された．

なお，これらのパターンは生成テストスイートによって網羅できない必要条件であるが十分条件ではない．すなわち，これらのパターンに該当する任意のメソッドが網羅できないコードを持つわけではない．

表 3: 生成テストスイートで網羅できない原因のパターン

パターン	件数
「特定の値を要求」	58
「特定の型を要求」	6
「実行不可能コード」	6
「マルチスレッド処理」	3

パターン「特定の値を要求」

パターン「特定の値を要求」とは，生成テストスイートで網羅できなかったコードを網羅するために，引数に何らかの条件を満たした値を必要とする場合を指す．生成テストスイートで網羅できない処理について，このパターンに分類されたメソッドで見られた処理の例を表 4 に示す．

表 4: 網羅のために引数に特定の値が必要な処理の例

網羅のために引数に特定の値が必要な処理	引数に要求される条件の例	条件を満たす引数の例
システムのプロパティを取得	`System.getProperty()` の正常系入力	`"user.dir"`
システムの環境変数を取得	`System.getenv()` の正常系入力	`"PATH"`
数値を表す文字列を数値に変換	`Float.parseFloat()` の正常系入力	`"10.0f"`
文字の種類（大文字や小文字等）を専用のメソッドで判定	`Character.isLowerCase()` の正常系入力	`'a'`
文字コードの大小関係で文字を比較	Unicode で `U+0x80` 以上 `U+0x800` 未満の文字	`'À'`
キャストでオーバーフローを判定	`long` 型では表現できるが `int` 型ではオーバーフローする整数	`2147483648L`

```
1 Float parseFloat(String value, float defaultValue) {
2     try {
3         return Float.parseFloat(value);
4     } catch (NumberFormatException e) {
5         ...
6     }
7 }
```

図 5: パターン「特定の値を要求」に該当するメソッドの例

```
1 int uncheckedIntCast(Object x) {
2     if (x instanceof Number)
3         return ((Number) x).intValue();
4     return ((Character) x).charValue();
5 }
```

図 6: パターン「特定の型を要求」に該当するメソッドの例

このパターンに該当するメソッドの例として，float 型の数値を表す文字列（"10.0f"など）を float 型の数値に変換するメソッドを図 5 に示す．このメソッドでは，3 行目の Float.parseFloat() の正常系処理が生成テストスイートによって網羅されていない．よって，この処理を網羅するためには，Float.parseFloat() に対して数値を表す文字列を入力するテストケース，すなわち，数値を表す文字列を引数に取るテストケースが必要である．

パターン「特定の型を要求」

パターン「特定の型を要求」とは，生成テストスイートで網羅できなかったコードを網羅するために，引数の型のクラスが持つ子クラスの型を要求する処理を含む場合を指す．

このパターンに該当するメソッドの例として，Object 型の引数を int 型にキャストしその値を返すメソッドを図 6 に示す．このメソッドでは，2 行目の条件の false 分岐，およびその分岐先の命令（4 行目）が網羅されていない．3 行目の分岐では，引数の型が Number 型かどうかを判定している．よって，4 行目を網羅するためには，Number 型以外の変数を引数に入力するテストケースが必要である．

パターン「実行不可能コード」

パターン「実行不可能コード」とは，引数にどのような値を与えても論理的に実行不可能なコードを含むため，生成テストスイートによる網羅ができない場合を指す．このパターンは，実行不可能な分岐方向と実行不可能な例外捕捉の 2 つに分類される．

実行不可能な分岐方向とは，引数にどのような値を与えても，ある一方の分岐方向しか実行できない場合を指す．このパターンに該当するメソッドの例を図 7 に示す．変数 ix は，3 行目において値 0 で

```
1  List<String> splitToList0(String str, char ch) {
2      List<String> result = new ArrayList<>();
3      int ix = 0, len = str.length();
4      for (int i = 0; i < len; i++) {
5          if (str.charAt(i) == ch) {
6              result.add(str.substring(ix, i));
7              ix = i + 1;
8          }
9      }
10     if ( ix >= 0 ) {
11         result.add(str.substring(ix));
12     }
13     return result;
14 }
```

図 7: 実行不可能な分岐方向を含むメソッドの例

```
1  String cast(Object value) {
2      if (value != null) {
3          try {
4              return value.toString();
5          } catch ( Exception e ) {
6          }
7      }
8      return null;
9  }
```

図 8: 実行不可能な例外捕捉を含むメソッドの例

初期化されており，ix の値が変化しうる 4〜9 行目の処理においても，その値が 0 から減少するような処理は存在しない．よって，10 行目の if 文の false 分岐は，引数にどのような値であっても実行不可能である．

実行不可能な例外捕捉とは，引数にどのような値を与えても，try-catch 文で指定された例外を捕捉できない場合を指す．このパターンに該当するメソッドの例を図 8 に示す．このメソッドでは，4 行目の value.toString() に対し，例外 Exception が try-catch 文によって捕捉されている．しかし，value.toString() は検査例外を投げないためこのメソッドの実行中に例外は発生しない．よって，5 行目の例外捕捉は実行不可能である．

パターン「マルチスレッド処理」

パターン「マルチスレッド処理」とは，生成テストスイートで網羅できなかったコードを網羅するために，マルチスレッド化したテストケースが必要な場合を指す．

RQ1 で調査したメソッドのうち，スレッドをスリープする処理を含むメソッドだけがこのパターンに分類された．そこで，このパターンに該当するメソッドの例として，そのような処理を含むメソッドを図 9 に示す．このメソッドでは，3 行目の Thread.sleep() から発生する検査例外

```
1 Integer apply(Integer i) {
2     try {
3         Thread.sleep(1);
4     } catch (InterruptedException e) {
5         e.printStackTrace();
6     }
7     return i;
8 }
```

図 9: パターン「マルチスレッド処理」に該当するメソッドの例

InterruptedException の捕捉（4 行目）と捕捉時の処理（5 行目）が生成テストスイートによって実行されていない．例外 InterruptedException は，スレッドが待ち状態，休止状態，占有されている状態のいずれかにおいて，スレッドに割り込みが発生した際に投げられる例外である．この例外を網羅するためには，マルチスレッド処理を利用して InterruptedException を意図的に発生させるテストケースが必要である．しかし，EvoSuite はマルチスレッド化したテストケースを生成できない [11] ため，この例外の発生を網羅できない．

7　RQ2 に対する調査

RQ1 の調査で得られたパターンに対し，生成テストスイートによって網羅できなかった原因を解決できるようにコードをリファクタリングする方法を調査した．

7.1　調査方法

RQ1 で得られたそれぞれの原因パターンに対し，以下の手順で調査を実施した．

1. 網羅できない原因を解決するリファクタリング手法を考案する
2. 原因パターンに該当するメソッドに対し考案手法でリファクタリングを実行する
3. リファクタリング後のメソッドに対し EvoSuite でテストを生成する
4. JaCoCo で網羅率を計測し，網羅できない原因が解決されたかどうか判定する

7.2　結果

RQ1 で得られたパターンのうち，「特定の値を要求」と「実行不可能コード」に該当する処理に対するリファクタリング方法を考案した．

パターン「特定の値を要求」

EvoSuite には，Java バイトコードに静的に埋め込まれたプリミティブ型および文字列の定数を参照する性質がある [2]．そこで，網羅できない原因を解決するリファクタリング方法として，メソッドの先頭に図 10 のような文を挿入する方法を考案した．この文は，もとのメソッドの処理に影響しないダミーコードである．このリファクタリング手法を適用することで，EvoSuite がテストスイート生成時にダミー判定に埋め込まれた定数を入力としてテストケースを生成するようになる．

この手法によって網羅できない原因を解決できたメソッドを説明する．6.2 節で例示した図 5 のメソッドをリファクタリングした結果を図 11 に示す．`Float.parseFloat()` を網羅できない原因は，`Float.parseFloat()` の入力として数値を表す文字列が要求されるためであった．そこで，このメソッドの先頭に，引数 `value` が数値を表す文字列の 1 つである`"10.0f"`と一致するかどうかを判定するダミーコードを挿入した．このメソッドに再び EvoSuite を使用してテストスイートを生成したところ，そのテストスイートには引数 `value` に`"10.0f"`を入力するテストケースが含まれていた．そのため，この手法によって網羅できない原因を解決できた．

一方で，以下の 3 つのすべての特徴を持つメソッドでは，この手法では網羅できない原因を解決できなかった．

```
1     if (1 番目の引数 == 条件を満たす定数 && 2 番目の引数 == 条件を満たす定数 && ...) {}
```

図 10: 挿入するダミー判定

```
1     Float parseFloat(String value, float defaultValue) {
2 +       if (value == "10.0f") {}
3         try {
4             return Float.parseFloat(value);
5         } catch (NumberFormatException e) {
6             ...
7         }
8     }
```

図 11: ダミー分岐の挿入により網羅できない原因を解決できたメソッドの例

1. 複数の引数を持つ
2. それぞれの引数が網羅に必要な条件を持つ
3. ある引数の条件が他の引数の条件に影響する

このメソッドで網羅できない原因を解決するには，すべての引数が網羅に必要な条件を満たした値を入力するテストケースが必要である．しかし，このケースに該当するいずれのメソッドにおいてもそのようなテストケースは生成されなかった．

このケースに該当するメソッドの例として図 12 のメソッドを示す．このメソッドは，文字列 `input` に含まれるホワイトスペースを `pos` の位置から探索し最初に発見した位置を返すメソッドである．このメソッドでは，6 行目の条件の `false` 分岐が網羅されていない．この分岐方向を網羅するには，以下の条件を満たした引数が必要である．

- 引数 `input`: 最初に登場するホワイトスペースより前にホワイトスペース以外の文字を含む
- 引数 `pos`: `input` で最初に登場するホワイトスペースより前の位置を指定する

この条件を満たす引数として，例えば，引数 `input` に`"aa bb"`という文字列を，引数 `pos` には `0` という値を入力する場合が該当する．そこで，図 12 の 2 行目のようにダミーコードを挿入し，再度 EvoSuite でテストスイートを生成した．しかし，`"aa bb"`と `0` の一方だけを入力するテストケースは生成されたが，これらを同時に入力するテストケースは生成されなかった．そのため，この手法では網羅できない原因を解決できなかった．

```
1     int skipWhitespace(String input, int pos) {
2 +       if (input == "aa bb" && pos == 0) {}
3
4         for (; pos < input.length(); pos++) {
5             char c = input.charAt(pos);
6             if ( c != ' ' && c != '\t' ) {
7                 break;
8             }
9         }
10        return pos;
11    }
```

図 12: ダミー分岐の挿入では網羅できない原因を解決できなかったメソッドの例

```
1     List<String> splitToList0(String str, char ch) {
2         List<String> result = new ArrayList<>();
3         int ix = 0, len = str.length();
4         for (int i = 0; i < len; i++) {
5             if (str.charAt(i) == ch) {
6                 result.add(str.substring(ix, i));
7                 ix = i + 1;
8             }
9         }
10 -       if (ix >= 0) {
11            result.add(str.substring(ix));
12 -       }
13        return result;
14    }
```

図 13: 実行不可能コードの削除によるリファクタリング

パターン「実行不可能コード」

網羅できない原因を解決するリファクタリング方法として，実行不可能コードを削除する手法を考案した．図 7 のメソッドをこの手法でリファクタリングした結果を図 13 に示す．6.2 節で述べたように，このメソッドの実行不可能コードは 10 行目の `ix >= 0` を `false` にする場合である．そこで，図 13 のように実行不可能コードを削除した．`ix >= 0` という分岐が網羅対象から外れるため，網羅できない原因が解決された．

8　妥当性の脅威

本研究では，5.2 節で述べたような特徴を持つメソッドを対象に調査した．しかし，実世界における Java プロジェクトでは，外部の変数やメソッドなどに依存したり，Java 標準パッケージ以外の外部パッケージと依存関係を持ったりするのが一般的である．よって，本研究とは異なるデータセットや Java プロジェクト等を対象に調査を実施した場合，異なる結果が得られる可能性がある．

本研究では，網羅基準として行網羅と分岐網羅に着目して調査した．しかし，EvoSuite は行網羅や分岐網羅以外にも，例外網羅や出力網羅といったさまざまな基準を網羅目標に設定する [9]．そのため，行網羅や分岐網羅以外に対して調査を実施した場合，異なる結果が得られる可能性がある．

本研究では，テスト生成ツールとして EvoSuite を使用した．2.4 節で述べたように，EvoSuite はテストスイートの生成に遺伝的アルゴリズムを使用する．そのため，生成されるテストスイートは，シード値や探索時間といったテスト生成時のパラメータによって変化する．よって，5 節の調査準備において，異なるパラメータを与えてテストスイートを生成した場合，以降の調査で異なる結果が得られる可能性がある．

また，2.4 節で述べたように，テスト生成ツールにはさまざまな種類が存在し，ツールによってテストの生成方針が異なる．そのため，Randoop や SUSHI といった EvoSuite 以外のツールを使用して調査を実施した場合，異なる結果が得られる可能性がある．

9　おわりに

本研究では，テスト生成ツールに入力するプログラムの構造が，生成テストスイートの網羅率に与える影響について調査した．調査準備として，Java メソッドに対し EvoSuite でテストスイートを生成し，行網羅と分岐網羅の少なくとも一方が 1.0 未満のメソッドを抽出した．その後，本調査として，まず，生成テストスイートが網羅できないコードに対し，そのコードを網羅できない原因を調査した．調査の結果，網羅できない原因として，「特定の値を要求」「特定の型を要求」「実行不可能コード」「マルチスレッド処理」の 4 つのパターンを発見した．次に，網羅できない原因を解決するようにコードをリファクタリングする手法を調査した．調査の結果，「特定の値を要求」に該当するメソッドの一部に適用可能なリファクタリング手法と，「実行不可能コード」に該当する全メソッドに適用可能なリファクタリング手法を考案した．

今後の課題としては，調査対象として，本研究で使用したデータセットとは異なるデータセットや，Randoop や SUSHI といった EvoSuite 以外のテスト生成ツールを用いた際に，どのような結果が得られるかを検証する点が挙げられる．

謝辞

本研究は，数多くの方の指導や支援なしには遂行できませんでした．

まず，指導教員の肥後芳樹教授には，一年間，研究テーマの決定から卒業論文の添削に至るまでさまざまな面で指導していただきました．毎週のグループミーティングや不定期の個別ミーティングではさまざまなアドバイスを頂き，私の研究活動を支援していただきました．

次に，楠本真二教授と柗本真佑助教には，中間報告などで貴重なご指摘をいただきました．そのおかげで，研究をより良い方向へと進められたと感じています．

事務補佐員の橋本美砂子さんには，研究室運営に関する活動はもちろんのこと，研究会への出張手続きやコーヒー会の飲料準備といった点においても助けとなりました．

楠本研究室の先輩方には，輪講でのアドバイスや文章の添削だけでなく，研究内容の相談から雑談に至るまで，ありとあらゆる面でお世話になりました．また，研究室旅行や忘年会，LT 会といった行事は，研究活動を行う上で精神的な励みとなりました．

同じく楠本研究室の同期には，研究という初めての活動にともに向き合う仲間として，あるいは雑談の話し相手として，さまざまな面で心の支えとなりました．ときには研究の相談を持ちかけたこともありましたが，そんなときでも親身に相談に乗ってくれました．

家族には，大学院への進学にかかる費用や，下宿生活における生活費といった金銭的な側面から多くの支援をしてもらいました．そのおかげで，生活に支障をきたすことなく安心して研究活動に取り組めたと感じています．

最後に，私の研究活動を支援してくださったすべての方に改めて感謝申し上げます．

参考文献

[1] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 2004.

[2] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[3] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *Proceedings of OOPSLA 2007 Companion*, 2007.

[4] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. SUSHI: A test generator for programs with complex structured inputs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, 2018.

[5] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 2010.

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 2005.

[7] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 2004.

[8] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. Evosuite at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021.

[9] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Search-Based Software Engineering*. IEEE Computer Society, 2015.

[10] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, and Kazuya Yasuda. Constructing dataset of functionally equivalent java methods. In *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022.

[11] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 2013.