

# 修士論文

ソースコードの類似性に基づいたテストコード  
自動推薦ツール SuiteRec

倉地 亮介

2020年1月28日

奈良先端科学技術大学院大学  
先端科学技術研究科 情報科学領域

本論文は奈良先端科学技術大学院大学先端科学技術研究科情報科学領域に  
修士(工学)授与の要件として提出した修士論文である。

倉地 亮介

審査委員 :

飯田 元 教授 (主指導教員)  
井上 美智子 教授 (副指導教員)  
市川 昊平 准教授 (副指導教員)  
崔 恩灝 准教授 (京都工芸纖維大学)

# ソースコードの類似性に基づいたテストコード 自動推薦ツール SuiteRec\*

倉地 亮介

## 内容梗概

ソフトウェアの品質確保の要と言えるソフトウェアテストを支援することは、重要である。これまでにテスト作成コストを削減するために、様々な自動生成技術が提案してきた。しかし、既存ツールによって自動生成されたテストコードは、テスト対象コードの作成経緯や意図に基づいて生成されていないという性質から保守作業を困難にさせる課題がある。この課題の解決方法として、既存テストの再利用が有効であると考えられる。本研究ではオープンソースソフトウェアに存在する品質が高いテストコードを推薦するツール SuiteRec を提案する。SuiteRec は、クローンペア間でのテスト再利用を考える。開発者からの入力コード片に対して類似コード片を検出し、その類似コード片に対応するテストスイートを推薦する。さらに、テストコードの良くない実装を表す指標であるテストスメルを開発者に提示し、より品質の高いテストスイートを推薦できるように推薦順位を並び替える。評価実験では、被験者によって SuiteRec を使用した場合とそうでない場合でテストコードの作成してもらい、テスト作成をどの程度支援できるかを定量的および定性的に評価した。その結果、SuiteRec を利用した場合、(1) 条件分岐が多いプログラムのテストコードを作成する際にコードカバレッジの向上に効果的であること、(2) 作成したテストコードはテストスメルの数が少なく品質が高いこと、(3) 開発者はテストコード作成作業を容易だと認識し、自分で作成したテストコードに自信が持てることが分かった。また、SuiteRec は開発者が参考にしたいテストスイートを上位に推薦できることを確認した。

---

\*奈良先端科学技術大学院大学 先端科学技術研究科 情報科学領域 修士論文、2020年1月28日。

## キーワード

コードクローン検出, 推薦システム, ソフトウェアテスト, テストスメル, 単体  
テスト

# **Automatic Test Suite Recommendation System**

## **based on Code Clone Detection\***

Ryosuke Kurachi

### **Abstract**

Software testing is vital to ensure the quality of software. In previous studies, various techniques to automatically generate test codes have been proposed to reduce development costs. However, automatically generated tests tend to ignore the development process and intention behind the target code, and therefore generally considered to be less readable and maintainable. This places a question mark over their practical value. To solve this problem, we propose **SuiteRec**, a recommendation tool to find existing high-quality test codes from OSS projects. The basic idea behind **SuiteRec** is that test codes can be reused between code clone pairs. **SuiteRec** finds code clones of the input code from OSS projects and recommends test suites corresponding to the clones. The recommended test suites are sorted by their quality and presented to the developer. Furthermore, test smells, which indicate bad implementation of test codes, are shown for each test suite. In the evaluation, we asked subjects to develop test codes with and without using **SuiteRec** and compared the developed test codes. We show that (1) **SuiteRec** improves code coverage when developing test codes for target codes with many conditional branches, (2) test codes created using **SuiteRec** have higher quality and less test smells, and (3) developers feel that it is easier to develop test codes, and they are more confident in the resulting test codes when using

---

\*Master's Thesis, Division of Information Science, Graduate School of Science and Technology, Nara Institute of Science and Technology, January 28, 2020.

**SuiteRec**. In addition, we have confirmed that **SuiteRec** can recommend test suites to the top rankings that developers want to refer.

**Keywords:**

clone detection, recommendation system, software testing, test smell, unit test

# 目 次

1. はじめに	1
2. 背景	3
2.1 コードクローン . . . . .	3
2.1.1 コードクローンの分類 . . . . .	3
2.1.2 コードクローン検出技術 . . . . .	3
2.2 ソフトウェアテスト . . . . .	4
2.2.1 テストの種類 . . . . .	5
2.2.2 テストスメル . . . . .	7
2.2.3 テストコード自動生成技術 . . . . .	12
2.2.4 既存の自動生成ツールにおける課題 . . . . .	13
3. SuiteRec: テストコード自動推薦ツール	15
3.1 Step1: 類似コード片の検出 . . . . .	16
3.2 Step2: テストコードの検索 . . . . .	17
3.3 Step3: テストスメルの検出 . . . . .	19
3.4 Step4: 推薦されるテストスイートの順位付け . . . . .	19
4. 評価実験	23
4.1 評価実験 1: テストコード作成支援に関する実験 . . . . .	23
4.1.1 評価実験 1 のデータセット . . . . .	24
4.1.2 評価実験 1 の手順 . . . . .	29
4.1.3 評価実験 1 の結果 . . . . .	31
4.2 評価実験 2: 推荐されるテストスイートの順位付けに関する実験 . . . . .	37
4.2.1 評価実験 2 のデータセット . . . . .	37
4.2.2 評価実験 2 の手順 . . . . .	39
4.2.3 評価実験 2 の結果 . . . . .	40
5. 議論	44

5.1 SuiteRec の有用性 . . . . .	44
5.2 妥当性への脅威 . . . . .	45
<b>6. 関連研究</b>	<b>47</b>
<b>7. まとめと今後の課題</b>	<b>48</b>
<b>謝辞</b>	<b>49</b>
<b>参考文献</b>	<b>50</b>
<b>付録</b>	<b>55</b>
<b>A. おまけその 1</b>	<b>55</b>
<b>B. おまけその 2</b>	<b>55</b>

## 図 目 次

1	テストにおけるタスク	5
2	Assertion Roulette の例	8
3	Default Test の例	9
4	Conditional Test Logic の例	10
5	Eager Test の例	10
6	Exception Handling の例	11
7	Mystery Guest の例	11
8	SBST によるテストケース生成の例	13
9	提案手法の概要	15
10	テストコードと対象コードの対応付け	18
11	SuiteRec から推薦されるテストコード	21
12	タスク 1 のプロダクションコード	25
13	タスク 2 のプロダクションコード	27
14	タスク 3 のプロダクションコード	28
15	命令網羅の平均カバレッジ	32
16	分岐網羅の平均カバレッジ	32
17	テストコード作成タスク終了までの時間	33
18	テストコード内に含まれていたテストスメルの数	34
19	実験後のアンケートの回答	36
20	タスク A のプロダクションコード	38
21	タスク B のプロダクションコード	39
22	おまけの図	55

## 表 目 次

1	テストの種類	6
2	タスクの割り当て	29
3	MAP と MRR の評価結果	42

4	MR@N の評価結果	42
---	------------	----

## 1. はじめに

近年，ソフトウェアに求められる要件が高度化・多様化する一方，ユーザからはソフトウェアの品質確保やコスト削減に対する要求も増加している[35]. その中でもソフトウェア開発全体のコストに占める割合が大きく，品質確保の要ともいえるソフトウェアテストを支援する技術への関心が高まっている[9]. しかし，現状ではテスト作成作業の大部分が人手で行われており，多くのテストを作成しようとするとそれに比例してコストも増加してしまう. このような背景から，ソフトウェアの品質を確保しつつコスト削減を達成するために，様々な自動化技術が提案されている[1, 10, 16, 22, 25].

既存研究で提案されている *EvoSuite*[10] は，単体テスト自動生成における最先端のツールである. *EvoSuite* は，対象コードを静的解析しプログラムを記号値で表現する. そして，対象コードの制御パスを通るような条件を集め，条件を満たす具体値を生成する. 単体テストを自動生成することで，開発者は手作業でのテスト作成時間が自動生成によって節約することができ，またコードカバレッジを大幅に向かうことができる. しかし，既存ツールによって自動生成されるテストコードは対象のコードの作成経緯や意図に基づいて生成されていないという性質から保守作業を困難にするという課題がある[7, 21, 23]. これは，自動生成ツールの実用的な利用の価値に疑問を呈する. テストが失敗するたびに，開発者はテスト対象のコード内で不具合の原因を特定するまたは，テスト自体を更新するか否かを判断する必要がある. 自動生成されたテストコードは読みづらく，開発者の助けになるというより，むしろ保守作業を困難にするという結果が報告されている[30]. 従って，自動生成技術によって一時的に削減できたコストは，後の保守作業を通して返済される可能性がある.

我々は，この課題の解決するために既存テストの再利用が有効であると考える. 本研究では，オープンソースソフトウェア(以下，OSS)に存在する既存の品質の高いテストコード推薦するツール *SuiteRec* を提案する. 推薦手法の基本となるアイディアはクローンペア間でのテストコード再利用である. *SuiteRec* は，入力コード片に対して類似コード片を検出し，その類似コード片に対応するテストスイートを開発者に推薦する. さらに，テストコードの良くない実装を表す指標で

あるテストスメルを開発者に提示し、より品質の高いテストスイートを推薦できるように推薦順位がランキングされる。

評価実験では、被験者によって SuiteRec の使用した場合とそうでない場合でテストコードの作成してもらい、テスト作成をどの程度支援できるかを定量的および定性的に評価した。その結果、SuiteRec の利用は条件分岐が多く複雑なプログラムのテストコードを作成する際に、コードカバレッジの向上に効果的であること、作成したテストコードに含まれるテストスメルの数が少なく品質が高いことが分かった。また、SuiteRec は開発者が参考にしたいテストコードを上位に推薦できることを確認した。実験後のアンケートによる定性的な評価では、SuiteRec を使用した場合、被験者はテストコードの作成が容易になると認識し、また自分の作成したコードに自信が持てることが分かった。

以降、2章では、本研究に関わるコードクローン及びソフトウェアテストについての背景を説明する。3章では、本研究で提案するテストコード自動推薦ツール SuiteRec について説明する。4章では、被験者による SuiteRec の評価実験について説明する。5章では、SuiteRec の有効性と妥当性への脅威について議論する。最後に6章で、まとめと今後の課題について説明する。

## 2. 背景

### 2.1 コードクローン

コードクローンとは、ソースコード中に存在する同一、あるいは類似した部分を持つコード片のことであり、コピーアンドペーストなどの様々な理由により生成される [36]. 互いにコードクローンになるコード片の対のことをクローンペアと呼び、クローンペアにおいて推移関係が成り立つコードクローンの集合のことをクローンクラスと呼ぶ. これまでの研究 [3, 5, 36] では、コードクローンの存在は、ソフトウェアの保守を困難にすると言われており除去すべきと考えられていた. しかし、最近の調査ではソフトウェアの開発・保守に影響を与えるのは一部のコードクローンだけであることが明らかになり、コードクローンを除去するのではなく活用した研究も多く提案されている.

#### 2.1.1 コードクローンの分類

既存研究 [14, 28] では、クローンペア間の違いの度合いに基づき、コードクローンを以下の 4 種類に分類している.

- タイプ 1：空白やタブの有無、括弧の位置などのコーディングスタイル、コメントの有無などの違いを除き完全に一致するコードクローン
- タイプ 2：タイプ 1 のコードクローンの違い加えて、変数名や関数名などのユーザ定義名、変数の型などが異なるコードクローン
- タイプ 3：タイプ 2 のコードクローンの違いに加えて、文の挿入や削除、変更などが行われたコードクローン
- タイプ 4：同一の処理を実行するが、構文上の実装が異なるコードクローン

#### 2.1.2 コードクローン検出技術

これまでの研究において、様々なコードクローン検出技術が提案されてきた [26, 36]. 本節では、代表的な 3 つの検出技術を紹介する.

### 文字/行単位の検出

この検出手法では、ソースコードを文字(または行)の並びで表現し、一定以上の長さで同じ文字(行)の並びが出現する部分をコードクローンとして検出する。他の検出技術に比べ検出速度が高速という利点があるが、文字列の並びを比較するため構文的に類似したコード片を検出するのが困難である。

### 字句単位の検出

字句単位の検出手法では、検出の前処理としてソースコードを字句の列に変換する。そして、閾値以上の長さで一致している字句の部分列をコードクローンとして検出する。文字/行単位の検出に比べ、構文的に類似したコード片に対しても、ある程度柔軟に検出することができる。

### 抽象構文木を用いた検出

抽象構文木とは、ソースコードの構文構造を木構造で表したグラフのことである。この検出手法では、検出の前処理としてソースコードに対して構文解析を行うことによって、抽象構文木を構築する。そして、抽象構文木上の同型の部分木をコードクローンとして検出する。他の検出技術に比べ、部分木の同型判定を行うので計算コストが高くなる。

## 2.2 ソフトウェアテスト

ソフトウェアテスト(以下、テスト)とは、ソフトウェア開発プロセスの中で最後の品質を確保する工程である。テストは、ソフトウェアが仕様通りに動作することを確認すること、また不具合を検出し修正することでソフトウェアの品質を向上させることを目的として行われる。テストは図1で示すように、テスト計画、テスト設計、テスト実行、テスト管理という大きく4つのタスクで構成される。テスト設計をさらに詳細に「テスト分析」、「テスト設計」、「テスト実装」のように分割する場合も存在するが、本研究ではテストケース(テスト項目)の作成に必要な作業をすべて「テスト設計」タスクとして扱う。テスト計画タスクでは、開発全体の計画に基づき、テスト対象、スケジュール、各タスクの実施体制・リソー

ス配分等の策定を行う。テスト設計タスクでは、設計書などソフトウェアの仕様が記述されたドキュメント等を基に、テストケースを作成する。テスト実行タスクでは、ソフトウェアを動作させ、それぞれのテストケースにおいてソフトウェアが期待通りの振る舞いをするかどうかを確認する。テスト管理タスクでは、テストの消化状況やソフトウェアの品質状況の確認を随時行い、テスト優先度やリソース見直しなどのアクションを行う。テスト工程のコスト削減のため、テスト実行タスクにおいて、単体テストでは JUnit<sup>1</sup>、結合テスト Selenium<sup>2</sup>、Appium<sup>3</sup> 等のテスト自動実行ツールの利用が進んでいる。しかし、テスト設計タスクは未だ手動で行うことが多く、自動化技術の実用化および普及が期待されている。



図 1 テストにおけるタスク

### 2.2.1 テストの種類

テスト工程は、表 1 のようにテスト対象の粒度によって単体テスト、結合テスト、システムテストの 3 種類に分類される。

単体テストは、クラスやメソッドを対象としたプログラムを検証するためのテストであり、テスト工程の中で最も小さい粒度のテストである。単体テスト設計タスクで作成されるテストケースは、テストプロセス、テスト入力値、テスト期待値から構成される。テストプロセスに従ってテスト対象のソフトウェアにテスト入力値を与え、その出力結果をテスト期待値と比較する。これが一致していればテストは合格となり、一致しなければ不合格となる。単体テスト設計タスクに

<sup>1</sup><https://junit.org/junit5/>

<sup>2</sup><https://selenium.dev/>

<sup>3</sup><http://appium.io/>

表 1 テストの種類

テスト粒度	説明
単体テスト	プログラムを構成する比較的小さな単位の部品が、個々の機能を正しく果たしているかどうかを検証するテスト
結合テスト	個々の機能を果たすプログラムの部品(単体)を組み合わせて、仕様通り正しく動作するかを検証するテスト
システムテスト	個々の機能や仕組みを総合した全体像のシステムとして、仕様通り正しく動作するかを検証するテスト

においては多くの場合、同値分割法、境界地分析法などのテストケース作成技法を用いてテスト入力値を作成するが、ソフトウェアの要求通りに動作するかを確認するために、多くのバリエーションのテスト入力値を作成する必要がある。

実行した単体テストが十分に行われているかを測定する基準の一つとして、コードカバレッジ(以下、カバレッジ)がある。カバレッジとは、テスト実行時にプロダクションコードが実行された割合を示す。カバレッジは、どのような基準で測定するかによって値が異なる。以下に本研究で扱う代表的な2つのカバレッジの測定方法を説明する。

### 命令網羅 (C0)

命令網羅 (C0) は、プログラム中に定義されたすべての「命令」について、1回以上実行されたか否かについて判定する測定方法である。Java 言語での「命令」は、ソースコードレベルでのステートメントを基準とする方法と、バイトコード上のインストラクション(実行命令)を基準とする方法の2つがある。どちらの場合でも命令網羅率が 100% であることは、単体テストでプログラム内の全命令を少なくとも 1 回は実行していることを示す。

### 分岐網羅 (C1)

分岐網羅 (C1) は、プログラム中の各分岐について、1回以上実行されたか否かについて判定する測定方法である。Java 言語での分岐とは if 文や switch 文などの条件分岐、try-catch 構文による例外処理などが相当する。分岐網

羅率が 100% であることは、単体テストでプログラムのすべての条件分岐が実行されていることを示す。

### 2.2.2 テストスメル

本節では、テストコードの品質を定量的に測定するための基準であるテストスメルについて説明する。

テストスメルとは、テストコードの良くない実装を示す指標である。プロダクションコードの設計だけでなく、テストコードを適切に設計することの重要性は元々 Beck ら [6] によって提唱された。さらに、Deursen ら [8] は 11 種類のテストスメルのカタログ、すなわちテストコードの良くない設計を表す実装とそれらを除去するためのリファクタリング手法を定義した。このカタログはそれ以降、19 個の新しいテストスメルを定義した Meszaros[19] によって拡張された。最近の研究では、これらのテストスメルがソフトウェア開発にもたらす影響について実証的な調査をしている。Bavota ら [4] は、18 のソフトウェアプロジェクトにおけるテストスメルの拡散及びソフトウェアの保守に対するテストスメルの影響を調査した。その結果、JUnit クラスの 82% が少なくとも 1 つのテストスメルの影響を受けていること、さらにテストスメルが存在するクラスは、開発者の理解や不具合特定に悪影響を与えること明らかにした。Tufano ら [32] は、ソフトウェアのライフサイクルにおけるテストスメルの重要性とその寿命を測定することを目的とした実証実験を行った。その結果、開発者はテストクラスに対して最初のコミットでテストスメルを導入し、およそ 80% のケースでテストスメルが除去されないことが分かった。Davide ら [31] は、テストスメルが存在するテストコードは、そうでないテストコードと比べて変更が多く実施され、テスト対象コードは不具合を含んでいる可能性が高いことを示した。また、テストスメルの存在は開発者のテストコードの理解に悪影響を与えるだけでなく、テストコードがプロダクションコード内の不具合を特定するのにあまり効果的でないと報告した。

以下に、現在提案されているテストスメルの内、本研究で扱う 6 種類のテストスメルを説明する。

- Assertion Roulette

- Default Test
- Conditional Test Logic
- Eager Test
- Exception Handling
- Mystery Guest

以降、それぞれのテストスメルについて説明する。

### Assertion Roulette

Assertion Roulette は、図 2 のようにテストメソッド内に複数の assert 文が存在する場合に発生する。各 assert 文は異なる条件をテストするが、開発者には各 assert 文のエラーメッセージは提供されない、そのため assert 文の 1 つが失敗した場合、失敗の原因を特定するが困難である。

```
@MediumTest
public void testCloneNonBareRepoFromLocalTestServer() throws Exception {
    Clone cloneOp = new Clone(false, integrationGitServerURIFor("small-
repo.early.git"), helper().newFolder());
    Repository repo = executeAndwaitFor(cloneOp);
    assertThat(repo, hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
    File readmeFile = new File(repo.getWorkTree(), "README");
    assertThat(readmeFile, exists());
    assertThat(readmeFile, ofLength(12));
}
```

複数のassert文が存在する

図 2 Assertion Roulette の例

### Default Test

Default Test は、図 3 のようにテストメソッド名が初期状態（意味のない名前）である場合発生する。ティスティングフレームを使用した場合、クラス・メソッド名が初期状態である。テストコードの可読性向上のために適切な名前に変更する必要がある。

```

public class ExampleUnitTest {
    @Test
        public void addition_isCorrect() throws Exception {
            assertEquals(4, 2 + 2);
        }

    @Test
    public void test01() throws InterruptedException {
        .....
        Observable.just(200)
            .subscribeOn(Schedulers.newThread())
            .subscribe(begin.asAction());
        begin.set(200);
        Thread.sleep(1000);
        assertEquals(beginTime.get(), "200");
        .....
    }
}

```

テストクラス名が初期状態のまま

メソッド名が初期状態のまま

図 3 Default Test の例

### Conditional Test Logic

Conditional Test Logic は、テストメソッド内に複数の制御文が含まれている場合に発生する(図4)。テストの成功・失敗は、制御フロー内にある assert 文に基づくのでテスト結果を予測できない。また、条件分岐が多く複雑なテストコードは可読性を下げる。

### Eager Test

Eager Test は、テストメソッド内でテスト対象クラスのメソッドを複数回呼び出す場合に発生する(図5)。1つのテストメソッドで複数のメソッドを呼び出すと、他の開発者は、どのテスト対象をテストするのか混乱が生じる。

### Exception Handling

Exception Handling は、テストメソッド内に例外処理が含まれている場合に発生する(図6)。例外処理は、対象コードに記述すべきで、テストコード内では正しく例外処理が行われるかを確認すべきである。

### Mystery Guest

Mystery Guest は、図7のようにテストメソッド内で外部リソースを利用

した場合に発生する。テストメソッド内だけでなく外部ファイルなど、外部リソースを使用すると見えない依存関係が生じる。何らかの影響で外部ファイルを削除されるとテストが失敗してしまう。

```
@Test
public void testSpinner() {
    String id = entry.getKey();
    Object resultObject = resultsMap.get(id);

    if (resultObject instanceof EventsModel) {
        EventsModel result = (EventsModel) resultObject;
        if (result.testSpinner.runTest()) {
            System.out.println("Testing " + id + " (testSpinner)");
            AnswerObject answer = new AnswerObject(entry.getValue(), "", new CookieManager(), "");
            EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application, answer);
            assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
            for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.get(i));
            }
        }
    }
}
```

複数の制御文が存在する

図 4 Conditional Test Logic の例

```
@Test
public void NmeaSentence_GPGSA_ReadValidValues(){
    NmeaSentence nmeaSentence = new
    NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,25,1.5,2.1*39");
    assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(), is("2.5"));
    assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(), is("1.3"));
    assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(), is("2.1"));
}
```

テスト対象コードのメソッド  
が複数回呼び出される

図 5 Eager Test の例

```

@Test
public void realCase() {
    Point p47 = new Point("47", 556612.21, 172489.274, 0.0, true);
    Abriss a = new Abriss(p34, false);
    a.removeDAO(CalculationsDataSource.getInstance());
    ...
    try {
        a.compute();
    } catch (CalculationException e) {
        Assert.fail(e.getMessage());
    }
    Assert.assertEquals("233.2435", this.df4.format(a.getMean()));
    Assert.assertEquals("43", this.df0.format(a.getMSE()));
    Assert.assertEquals("30", this.df0.format(a.getMeanErrComp()));
}

```

テストメソッド内に例外  
処理の記述が存在する

図 6 Exception Handling の例

```

public void testPersistence() throws Exception {
    File tempFile = File.createTempFile("systemstate-", ".txt");
    try {
        SystemState a = new SystemState(then, 27, false, bootTimestamp);
        a.addInstalledApp("a.b.c", "ABC", "1.2.3");

        a.writeToFile(tempFile);
        SystemState b = SystemState.readFromFile(tempFile);

        assertEquals(a, b);
    } finally {
        //noinspection ConstantConditions
        if (tempFile != null) {
            //noinspection ResultOfMethodCallIgnored
            tempFile.delete();
        }
    }
}

```

外部にファイルを生成し、  
テストプロセスで利用している

図 7 Mystery Guest の例

### 2.2.3 テストコード自動生成技術

テスト工程の支援するために様々なテストコード自動生成技術が提案されている。既存の研究[17]は、既存のテストケースを再利用、自動生成、または再適用することによって、ソフトウェア開発のテスト工程における時間とコストを大幅に節約できることを示している。テスト生成技術は、主にランダムテスト(RT)、記号実行(SE)、サーチベーステスト(SBST)、モデルベース(MBT)、組み合わせテストの5つに分類できる。SEはさらに静的記号実行(SSE)と動的記号実行(DSE)に分けられる。

既存研究で提案されているEvoSuite[10]は、単体テスト自動生成における最先端のツールである。EvoSuiteは、SBSTを実装したツールであり、2011年に発表されて以来EvoSuiteをベースとした数多くの研究がなされている。SBSTでは、一般的に以下の手順でテストスイート(テストケースのまとまり)を生成する。

1. 達成したい要件に対する達成度合いを定量的に評価できる評価関数を設計
2. 予め用意したテストスイートをテスト対象に対して実行し、実行したテストスイートの評価関数の値を取得
3. 取得した評価関数の値が優れているテストスイートを元に、ヒューリスティック探索アルゴリズムによって新規にテストスイートを生成
4. 3で生成したテストスイートをテスト対象に対して実行し、実行したテストスイートの評価関数の値を取得
5. 設定した探索打ち切り条件を満たすまで、3, 4を繰り返し実行

評価関数の設計方法は、テスト実施の観点によって異なる。例えば、SBSTを用いてコードカバレッジ向上を目指したテストを実施する場合、評価関数は分岐網羅率等が用いられる。

SBSTを用いたテストケース自動生成の例を提示する。図8において、SBSTを用いて分岐1で「 $y > 1$ 」を満たすようなテスト入力値を生成したい場合、評価関数をプログラムが分岐1に到達したタイミングで評価する。このとき、 $x$ の値

が大きいほど評価関数の値も大きくなり、「 $y > 1$ 」を満たす度合いが大きくなると定量的に評価することができる。まず、 $x = -10$ として実行すると、評価関数の値は、 $E = -10$ となる。続いて、仮に $x = -5$ として実行すると、評価関数の値は、 $E = -5$ となる。この場合、後者のテスト入力値の方が「 $y > 1$ 」を達成するためには優れているテスト入力値、つまり $x$ の値が大きいテスト入力値をベースに、新しいテスト入力値の生成が行われる。それにより徐々に $x$ の値が大きいテスト入力値が生成されていき、最終的に $x = 1$ 等のテスト入力値が取得できる。

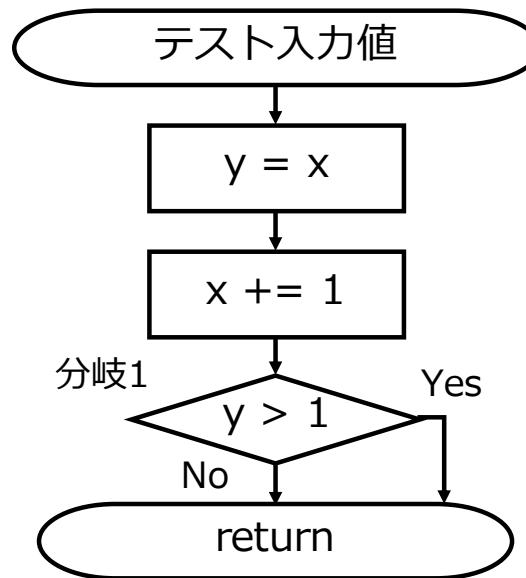


図 8 SBST によるテストケース生成の例

#### 2.2.4 既存の自動生成ツールにおける課題

2.2 節では、既存の単体テスト自動生成ツール EvoSuite について説明した。単体テストを自動生成することで、開発者は手作業でのテスト作成時間を節約することができ、またコードカバレッジを大幅に向上することができる。しかし、既存ツールによって自動生成されたテストコードは、対象コードの作成経緯や意図に基づいて生成されていないという性質から可読性が低く開発者に信用されてい

ないことや、後の保守作業を困難にするという課題がある [7, 21, 23]. このことは、自動生成ツールの実用的な利用の価値に疑問を呈する. テストが失敗するたびに、開発者はテスト対象のコード内で不具合の原因を特定するまたは、テスト自体を更新するか否かを判断する必要がある. このようにして、自動生成技術によって一時的に削減できたコストは、後の保守作業を通して返済される可能性がある [30].

Palomba ら [20] は、既存ツールによって自動生成されるテストコードは、プロジェクト内にテストスメルを拡散させることを確認した. 自動生成された JUnit クラス内 83%が、少なくとも 1 つのテストスメルの影響を受けることを報告した. また、自動生成されたテストコード内では “Assertion Roulette”, “Eager Test”, “Test Code Duplication” の 3 つのテストスメルが多く検出される特徴があり、いくつかのテストスメルは同時に存在していることが明らかにした.

自動生成ツールによって大量に生成されるテストコードは、プロジェクト内にテストスメルを拡散させ、開発者の可読性と保守性に大きな影響を与える可能性がある. 一般にソフトウェアの保守にかかる継続的なコストは、テストコード作成コストをはるかに上回るため、初めに理解しやすく良質なテストコードを作成する必要がある.

我々は、この課題の解決するために既存テストの再利用が有効であると考える. 本研究では、OSS に存在する既存の品質の高いテストコード推薦するツールを提案する. 既存テストの利用は、コーディング規約や命名規則に従った可読性の高いテストコードを利用できることや、人によって作成された信頼性の高いテストコードを利用できると考える.

### 3. SuiteRec: テストコード自動推薦ツール

本章では、本研究で提案するコードクローン検出技術を用いたテストコード自動推薦ツール SuiteRec について述べる。本研究では、コードクローン検出技術を利用し、OSS 上に存在する既存の高品質なテストコード推薦することで、開発者のテストコード作成を支援することが目的である。SuiteRec の基となるアイディアは、クローンペア間でのテストコード再利用である。SuiteRec は、開発者からの関数単位の入力コード片に対してその類似コード片を検出する。そして、類似コード片に対応するテストスイートを開発者に推薦する。さらに、推薦されるテストスイート内に含まれるテストスメルを提示し、より品質が高いテストスイートを推薦できるように推薦順位がランキングされる。

図 9 は、SuiteRec によってテストスイートが推薦されるまでの流れを示す。推薦手法は、主に以下の 4 つのステップから構成される。

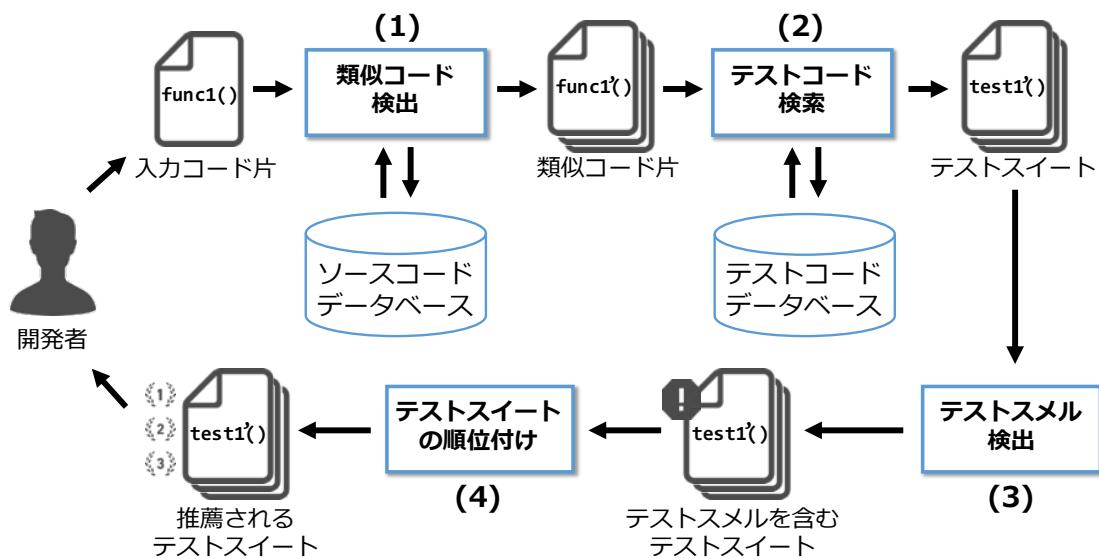


図 9 提案手法の概要

## Step1

SuiteRec は、開発者から入力コード片を受け取ると、既存のコードクローン検出ツールを用いて、入力コード片に対応する類似コード片を検出する。

## Step2

複数の類似コード片が検出されると、次にその類似コード片に対応するテストスイートをテストコードデータベース内から検索する。

## Step3

各類似コード片のテストスイートが検出されると、次にそれらをテストスメル検出ツールにかけ、各テストスイートに含まれるテストスメルを検出する。

## Step4

最後に、Step1 で得られた類似コード片と入力コード片の類似度と Step3 で検出されたテストスメルの数を基に出力されるテストスイートの順番がランギングされる。

以降の節で、各ステップの詳細について説明する。

### 3.1 Step1: 類似コード片の検出

Step1 では、開発者から受け取った関数単位のコード片に対して類似コード片を検出する。本研究では、コードクローン検出ツールとして NiCad[27] を採用した。NiCad は検出対象のコード片のレイアウトを統一的に変換させ、行単位で関数単位のコード片を比較することで、クローンペアを検出するツールであり、このような手法を取ることで、高精度・高再現率でのクローンペアの検出を実現している。

本研究で、NiCad を採用した理由として以下の 2 点が挙げられる。

- 単体テストの再利用を考える上で関数単位のコード片を高精度で検出できる
- 構文的に類似した type2, type3 のコードクローンを高速に検出できる

NiCad は、プロジェクトを入力とし、プロジェクト内のコードクローンをすべて検出する。検出されるコードクローンはクローンクラスとして出力される。SuiteRec は、NiCad の検出結果の中で入力コード片を含むクローンクラスを特定し、そのクローンクラスに含まれるコード片をすべて類似コード片として扱う。

SuiteRec は、NiCad を用いて入力コード片に対応する類似コード片を大規模な OSS プロジェクトを保持する Github<sup>4</sup>のリポジトリから検索する。図9のソースコードデータベースには、テストコードが存在する Github 上の 3,205 プロジェクトのプロダクションコードが格納されている。具体的には、既存のコード検索エンジンで利用されているデータセット [15] の内、テストフォルダが存在し、JUnit のティスティングフレームワークを採用しているプロジェクトを選択した。

NiCad は、一度に検索できるプロジェクトの規模限度がある。我々は、検索時間を短縮するために検索対象のプロジェクトに前処理をした。具体的には、大規模なプロジェクトは分割し、小規模なプロジェクトは統合させた状態で検索処理を実行した。さらに、検索処理を複数のプロジェクトに対して並列して走らせることで、現実的な時間での類似コードの検索を実現した。また、検出設定については NiCad の標準設定で SuiteRec に実装した。

### 3.2 Step2: テストコードの検索

Step2 では、Step1 で検出された類似コード片に対応するテストコードを検索する。大規模なプロジェクトのテストコードが格納されているテストコードデータベース(以下、TDB)からテストコードを検出するために、テスト対象コードとテストコードの対応付けを行う。

本研究では、対象コードとテストコードを対応付けるために以下の 3 つのフェーズを実施する。

1. 命名規則によるクラス単位での対応付け
2. テストコードを静的解析し、各テストケースから呼び出されるすべてのメソッド名を収集する

---

<sup>4</sup><https://github.com/>

3. テストメソッド名を区切り文字や大文字で分割し、対象メソッドと部分一致した場合、テストコードと対象コードをメソッド単位で対応付ける

本研究では、テストコードとして JUnit テスティングフレームワークを用いた単体テストを対象とする。フェーズ 1 では、JUnit の命名規則に従ってテストクラス名の先頭または、末尾に “Test” という文字列が含まれるテストクラスを収集し、収集したテストクラスから “Test” を除いたクラス名をテスト対象クラスとする。例えば、Calculator クラスと CalculatorTest クラスが対応付けられる。

フェーズ 2 では、テストコードを静的解析し、テストメソッド内で呼び出されるメソッド名を取得する。単体テストでは、図 3 の例のようにテストコード内でオブジェクトの生成を行い、テスト対象コードのメソッド呼び出して実行される。すなわち、TCD 内のテストコードを静的解析し、メソッド呼び出しを取得することで、テスト対象コードとテストコードを対応付ける。ただし、テストメソッド内では、複数のメソッドが呼び出されていることも考えられるので、フェーズ 3 では、さらにメソッド名の比較も行う。テストメソッド名の記述方法としてテスト対象メソッドの処理の内容を忠実に表すことが推奨されており、対象メソッドの名前が記述されていることが多い [2]。従って、テストメソッドの名前を区切り文字や大文字で分割し、対象メソッドと部分一致した場合、対応付けるように実装した。

<pre>public class Calculator {     public int multiply(int x, int y) {         return x * y;     } }</pre>	 <pre>@Test public void testMultiplyOfTwoNumbers() {     Calculator calc = new Calculator();     int expected = 200;     int actual = calc.multiply(10,20);     assertEquals(expected,actual); }</pre>
テスト対象コード	テストコード

図 10 テストコードと対象コードの対応付け

### 3.3 Step3: テストスメルの検出

Step3 では、Step2 で検索されたテストスイート内のテストスメルを検出する。本研究では、テストスメル検出ツールとして tsDetect<sup>5</sup>[24] を採用した。tsDetect は AST ベースの検出手法で実装されたツールであり、19 種類のテストスメルを検出できるツールである。また、85%～100% の精度と 90%～100% の再現率でテストスメルを検出できる。tsDetect は、既存研究で利用実績があり高精度で多くのテストスメルを検出できるので本研究でも利用した。本研究では、tsDetect で検出できる 19 種類のテストスメルの内、2.2.2 節で説明したテストコードの推薦を考える上で重要な 6 種類のテストスメルを提示するように実装した。

tsDetect は、AST ベースの検出手法であり大規模なテストコードに対して実行すると計算コストが高い。我々は、事前に TDB 内のテストコードのテストスメルを検出しその情報をテストコードに対応付けて TDB に格納した。これにより、推薦プロセスにおけるテストスメル検出時間を短縮化した。また、推薦テストコードとして相応しくない以下の 4 つのテストスメルを含むテストコードを事前に TDB から除去し、推薦されるテストコードとして出力されないようにした。

- **Empty Test** : テストメソッド内にテストの記述が無く、コメントのみが含まれているテストコード
- **Ignored Test** : @Ignore アノテーションがあり、実行されないテストコード
- **Redundant Assertion** : 必ずテストが成功する意味のないテストコード
- **Unknow Test** : assert 文が存在しないテストコード

### 3.4 Step4: 推薦されるテストスイートの順位付け

最後の Step4 では、推薦されるテストスイートを順位付けし、開発者が参考にしたいテストスイートを上位に提示できるようにテストスイートの並び替えを行う。

---

<sup>5</sup><https://testsmells.github.io/>

`SuiteRec` の順位付けは、Step1 の入力コード片と類似コード片の類似度と Step3 で検出されたテストスメルの数を基に計算する。我々は、以前の調査 [34] で、クローンペア間とテストコードの類似度の関係を調査した。詳細には、OSS 上に存在する 3 つの有名 Java プロジェクト内にある両方のコード片にテストコードが存在するコードクローニングを対象に、クローンペア間の類似度とそれぞれのコード片に対応するテストコードの類似度の関係を調査した。その結果、テストコード間の類似度と対象のクローンペア間の類似度には相関関係があり、クローンペア間の類似度が高いほどテストコードを再利用できる可能性が高いことが分かった。`SuiteRec` では、この結果を基に類似度が高いクローンペアの順に並び替え、さらに類似度が同じ場合、テストスメルの数で推薦するテストスイート順位を決定するように推薦ランキングを実装した。

本研究では、クローンペア間の類似度を NiCad で用いられる計算方法である *Unique Percentage of Items* (以下、*UPI*) を利用して計算する。以下に、*UPI* の計算式を示す。

$$\text{Unique Percentage of Items} (\text{UPI}) = \frac{\text{No. of Unique Items} * 100}{\text{Total No. of Items}}$$

ここで、*No. of Unique Items* は、入力コード片と類似コード片を行単位で比較した際に一致した行の数を意味する。*Total No. of Items* は、コード片内のすべての行の数を意味する。比較対象のコード片の *Total No. of Items* が異なる場合、行数が大きい方の *Total No. of Items* を用いて計算する。例えば、入力コード片の行数が 10、類似コード片の行数が 12 で、一致した行数が 9 の場合、入力コード片と類似コード片の類似度は 75% になる。

**Clone Pairs 1 : 71.4%**

Input Code		Similarity Code			
<pre> public String fizzBuzz(int number) {     if (number &lt;= 0) {         return "Not Natural Number";     }     if (number % 15 == 0) {         return "FizzBuzz";     } else if (number % 3 == 0) {         return "Fizz";     } else if (number % 5 == 0) {         return "Buzz";     } else {         return Integer.toString(number);     } } </pre>		<pre> public String fizzBuzz(int number) {     if (number &lt;= 0) {         throw new RuntimeException();     }     if (number % 15 == 0) {         return "FizzBuzz";     } else if (number % 3 == 0) {         return "Fizz";     } else if (number % 5 == 0) {         return "Buzz";     } else {         return Integer.toString(number);     } } </pre>			
Test Suite					
Assertion Roulette	Conditional Test Logic	Default Test	Eager Test	Exception Handling	Mystery Guest
<p>Lines 8 - 13 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java</p> <pre> @Test(expected=RuntimeException.class) public void test1() {     FizzBuzz1 fb = new FizzBuzz1();     fb.fizzBuzz(-1); } </pre> <p>Lines 14 - 21 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java</p> <pre> @Test public void returnBuzz_input5() throws Throwable {     FizzBuzz1 fizzBuzz = new FizzBuzz1();     String actual = fizzBuzz.fizzBuzz(5);     String expected = "Buzz";     assertEquals(expected, actual); } </pre> <p>Lines 22 - 29 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java</p> <pre> @Test public void returnNum_input2() throws Throwable {     FizzBuzz1 fizzBuzz = new FizzBuzz1();     String actual = fizzBuzz.fizzBuzz(2);     String expected = "2";     assertEquals(expected, actual); } </pre> <p>Lines 30 - 37 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java</p> <pre> @Test public void returnFizz_input3() throws Throwable {     FizzBuzz1 fizzBuzz = new FizzBuzz1();     String actual = fizzBuzz.fizzBuzz(3);     String expected = "Fizz";     assertEquals(expected, actual); } </pre> <p>Lines 38 - 45 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java</p> <pre> @Test public void returnFizzBuzz_input15() throws Throwable {     FizzBuzz1 fizzBuzz = new FizzBuzz1();     String actual = fizzBuzz.fizzBuzz(15);     String expected = "FizzBuzz";     assertEquals(expected, actual); } </pre>					

図 11 SuiteRec から推薦されるテストコード

図 11 は SuiteRec から推薦されるテストコードの例である。以下に SuiteRec のインターフェースについて各項目の説明を示す。

- (1) **Input Code Fragment.** 開発者が入力した関数単位のプロダクションコードが表示される。
- (2) **Similarity Code Fragment.** 入力コードに対する類似コードが表示される。入力コードと類似度コードの違いが分かるように差分がハイライトされる。
- (3) **Degree of similarity.** 入力コードと類似コードの類似度が表示される。類似度は NICAD で用いられている計算方法 Unique Percentage of Items (UPI) を採用した。
- (4) **Test Smells.** 推薦されるテストスイート内にテストスメルが含まれている場合、そのテストスメルがオレンジ色にハイライトされ開発者にテストスメルの存在を提示する。
- (5) **Recommend Test Suites.** 推薦されるテストスイートが表示される。また、どのプロジェクトからテストコードが参照されたのかを示すためにファイルパスも表示される。

## 4. 評価実験

この章では, SuiteRec を定量的及び定性的に評価するために, 被験者による評価実験について説明する. 評価実験は, 主に以下の 2 つの実験を実施した.

- **評価実験 1 :** テストコード作成支援に関する実験
- **評価実験 2 :** 推薦テストコードの順位付けに関する実験

以降, 各評価実験の詳細について説明する.

### 4.1 評価実験 1: テストコード作成支援に関する実験

評価実験 1 では, SuiteRec が開発者のテストコードをどの程度支援できるかを評価する. 被験者に 3 つのプロダクションコードのテストコードを作成してもらい, SuiteRec を使用して作成した場合とそうでない場合のテストコードを比較することで評価を行う. 実験を通してコードカバレッジ, 実験タスクを終了するまでの時間及びテストコードの品質に関するデータを収集することで, 以下のリサーチクエスチョンに答えることを目指す.

**RQ1:** SuiteRec は高いカバレッジを持つテストコードの作成を支援できるか?

RQ1 では, 被験者が作成したテストコードのカバレッジを測定する. テスト工程において, ソフトウェアの品質を確認する 1 つの指標としてカバレッジは重要な要素である. テストコード内で一度も実行されない行が存在するとその部分の品質を確保することはできない. SuiteRec の利用は高いカバレッジを達成するために役に立つか?

**RQ2:** SuiteRec はテストコードの作成時間を削減できるか?

RQ2 では, 被験者のテストコード作成タスクに費やす時間を測定する. 開発者は SuiteRec で推薦されるテストコードを参考にすることで, テストコード作成時間を短縮化できるのか?

**RQ3:** SuiteRec はテストスメルの数が少ないテストコードの作成を支援できるか？

RQ3 では、被験者が作成したテストコード内に含まれるテストスメルを検出し、テストコードの品質を測定する。開発者は SuiteRec で推薦されるテストコードを参考にすることで、品質の高いテストコードを作成することができるのか？

**RQ4:** SuiteRec の利用は、開発者のテストコード作成タスクの認識にどう影響するか？

RQ4 では、テストコード作成タスクを完了した被験者に作成タスクに関するアンケート回答してもらった。SuiteRec を利用した場合、テストコードの作成が容易になり、自身で作成したテストコードに自信が持てるのか？

以降、4.1.1 節では、評価実験 1 で用いた実験用データセットについて説明する。4.1.2 節では、評価実験 1 の手順について説明する。最後に 4.1.3 節で、評価実験 1 の結果について説明する。

#### 4.1.1 評価実験 1 のデータセット

評価実験 1 を実施するために、基本的なプログラミングスキルを保有し、ソフトウェアテストに関する知識を持つ学生を募集した。そして、情報科学を専攻するの修士過程の学生 10 人に対して実験を実施した。事前アンケートによると 9 割の学生が 2 年以上のプログラミング経験があり、8 割の被験者が 1 年以上の Java 言語の経験があった。また、すべての学生が授業などの講義でソフトウェアテストに関する基本的な知識を持っており、8 割が単体テストの作成経験があった。

評価実験 1 では、被験者に 3 つの実験タスクを割り当てた。以降、これら実験タスクをそれぞれ、タスク 1、タスク 2、タスク 3 と呼ぶ。被験者はテストコードを作成するのでプロダクションコードの仕様を十分理解していることが前提になる。そこで、我々はプロダクションコードとして競技プログラミングをよく用いられる典型的な計算問題を実験タスクとして選択した。また、各タスクの仕様を確認できるように自然言語で記述された仕様書を用意した。3 つの各タスクで違いを出すためにタスク 1、2、3 の順に条件分岐の数を 8、16、24 と多くなるように設定した。以下に、各タスクの概要を示す。

## タスク 1

タスク 1 は、典型的なFizzBuzz の Java メソッドに対するテストコードを作成するタスクである。このタスクは、被験者が一番始めに実施するタスクであり、条件分岐の数が 8 で 3 つのタスクの中で最も単純な構造のプログラムである。以下に、タスク 1 のプロダクションコードの仕様とソースコードを示す。

### タスク 1 のプロダクションコードの仕様

整数を入力し、次の条件で結果を返すプログラム。

- 3 で割り切れる場合、 “fizz” を返す。
- 5 で割り切れる場合、 “buzz” を返す。
- 3 と 5 両方で割り切れる場合、 “fizzbuzz” を返す。
- 入力が自然数でない場合、 “Not Natural Number” を返す。
- 上記以外は、入力数値を String 型で返す。

```
public class Experiment01 {  
  
    public String fizzBuzz(int number) {  
        if (number <= 0) {  
            return "Not Natural Number";  
        }  
        if (number % 15 == 0) {  
            return "fizzbuzz";  
        } else if (number % 3 == 0) {  
            return "fizz";  
        } else if (number % 5 == 0) {  
            return "buzz";  
        } else {  
            return Integer.toString(number);  
        }  
    }  
}
```

図 12 タスク 1 のプロダクションコード

## タスク 2

タスク 2 は、第 1 引数に応じて残り 3 つの引数の計算方法を変更し、結果を出力する Java メソッドのテストコードを作成するタスクである。このタスクは、条件分岐の数が 16 でタスク 1 より比較的複雑な構造のプログラムである。以下に、タスク 2 のプロダクションコードの仕様とソースコードを示す。

### タスク 2 のプロダクションコードの仕様

第 1 引数に応じて残り 3 つの引数の計算方法を変更し、結果を返す。

- 第 1 引数が “Medium” の場合、残り 3 引数の中央値を返す。
- 第 1 引数が “max” の場合、残り 3 引数の最大値を返す。
- 第 1 引数が “min” の場合、残り 3 引数の最小値を返す。
- 第 1 引数が上記以外の場合、 -1 を返す。

## タスク 3

タスク 3 は、2 つの int 型の値を入力し、試験の合否を判定するプログラムのテストコードを作成するタスクである。このタスクは、条件分岐の数が 24 個存在し、3 つのタスクの中で最も複雑な構造のプログラムである。以下に、タスク 3 のプロダクションコードの仕様とソースコードを示す。

```
public class Experiment02 {

    public int calcMediumMinMax(String select,int a, int b, int c)
    {
        if (select == "Medium"){
            if (a < b) {
                if (b < c) {
                    return b;
                } else if (a < c) {
                    return c;
                } else {
                    return a;
                }
            } else {
                if (a < c) {
                    return a;
                } else if (b < c){
                    return c;
                } else {
                    return b;
                }
            }
        }
        else if (select == "max"){
            return Math.max(a, Math.max(b,c));
        }
        else if (select == "min"){
            return Math.min(a, Math.min(b,c));
        }
        else {
            return -1;
        }
    }
}
```

図 13 タスク 2 のプロダクションコード

### タスク 3 のプロダクションコードの仕様

2つのスコア(0~100点)を入力し、次の条件に従って合格・不合格を判定するプログラム。

- スコアが0~100点以外の場合“Invalid Input”を返す。
- どちらかのスコアが0点であれば“failure”を返す。
- 両方とも60点以上の場合“pass”を返す。
- 合計が130点以上の場合“pass”を返す。
- 合計が100点以上で、どちらかのスコアが90点以上であれば“pass”を返す。
- 上記以外は“failure”を返す。

```
public class Experiment03 {  
    public String returnResult(int score1, int score2){  
        if(( score1 < 0 || score2 < 0 ) || ( score1 > 100 || score2 > 100 )) {  
            return "Invalid Input";  
        }else if( score1 == 0 || score2 == 0 ){  
            return "failure";  
        }else if( score1 >= 60 && score2 >= 60 ){  
            return "pass";  
        }else if( ( score1 + score2 ) >= 130 ) {  
            return "pass";  
        }else if( ( score1 + score2 ) >= 100 && ( score1 >= 90 || score2 >= 90 ) ) {  
            return "pass";  
        }else {  
            return "failure";  
        }  
    }  
}
```

図 14 タスク 3 のプロダクションコード

#### 4.1.2 評価実験 1 の手順

4.1 節で説明した実験タスクを用いた評価実験 1 の手順について説明する。まず、被験者間の事前知識の違いによる結果の相違を無くすために、実験前にソフトウェアテストに関する基本的な知識から JUnit の使用に関する 30 分の講義を実施した。また、被験者に SuiteRec の使い方を説明し、実際に練習問題で使用してもらいツールの利用とテストコードの作成について理解していることを確認した。

その後、用意した 3 つの実験タスクに対してテストコードを作成してもらった。被験者には、与えられた 3 つタスクを SuiteRec を使用した場合とそうでない場合でテストコードを作成してもらった。本実験では、タスクの終了は被験者に判断してもらう。具体的には、被験者自身が作成したテストコードのカバレッジ・品質に満足したとき、実験タスクを終了してもらった。実験時間は 1 つのタスクにつき最大 25 分の時間を設けた。我々は、SuiteRec の利用効果がタスクによって偏らないように、図 2 のように被験者を 2 つのグループに分け、グループによって SuiteRec の利用の有無をタスクによって変えるように割り当てた。また、SuiteRec を利用した場合の学習効果を防ぐために、3 つのタスクで連続して SuiteRec を利用しないようにタスクの割り当てを行った。さらに実験中、被験者は過去の回答を参考できないようにした。

表 2 タスクの割り当て

グループ	A		B	
	タスク	ツール	タスク	ツール
1 回目	タスク 1		タスク 1	○
2 回目	タスク 2	○	タスク 2	
3 回目	タスク 3		タスク 3	○

最後に、実験タスク終了後に被験者にテストコード作成に関するアンケートに答えてもらった。以下に、実施したアンケートの項目を示す。被験者は、これらのアンケート項目に対して 5 段階評価(強く反対・反対・どちらでもない・賛成・強く賛成)で回答してもらった。

評価実験 1: 実験タスクに関するアンケート項目

- Q1.** 今回の実験内容(課題)を理解できました。
- Q2.** 実験タスクを終えるのに十分な時間がありました。
- Q3-a.** テストコードの作成は簡単でした(ツール使用)。
- Q3-b.** テストコードの作成は簡単でした(ツール不使用)。
- Q4-a.** 作成したテストコードのカバレッジに自信がある(ツール使用)。
- Q4-b.** 作成したテストコードのカバレッジに自信がある(ツール不使用)。
- Q5-a.** 作成したテストコードの品質に自信がある(ツール使用)。
- Q5-b.** 作成したテストコードの品質に自信がある(ツール不使用)。
- Q6.** 推薦ツールの使用はテストコードを作成する際に参考になりました。

#### 4.1.3 評価実験 1 の結果

本節では、評価実験 1 の被験者による SuiteRec の定量的および定性的評価の結果を報告する。本章の前半で説明した、4つのリサーチクエスチョンについての分析結果を提示する。

**RQ1:** SuiteRec は高いカバレッジを持つテストコードの作成を支援できるか？

RQ1 では、SuiteRec を使用した場合とそうでない場合の被験者が、作成したテストコードのカバレッジを比較する。本実験では、被験者によって提出されたテストコードの命令網羅と分岐網羅の 2 種類のカバレッジを計算した。カバレッジの計算には、統合開発環境 Eclipse<sup>6</sup> のプラグインとして搭載されている EclEmma<sup>7</sup> を利用した。図 15 と図 16 はそれぞれ被験者による命令網羅と分岐網羅の平均カバレッジを示す。結果として、命令網羅の割合は 3 つのタスクすべてにおいてツールを利用した場合とそうでない場合で網羅率にほとんど違いはなく、どのタスクも網羅率が 90% を超えている。図 16 の分岐網羅についても分岐数が少ないタスク 1 とタスク 2 については、ツールを使用した場合とそうでない場合でほとんど差がないことが分かる。しかし、プロダクションコードの分岐数が最も多いタスク 3 については、実験者の平均カバレッジに 10% 以上の差があることが分かった。この結果は、分岐が多いプロダクションコードのテストコードを作成する際に、SuiteRec で推薦されるテストコードは、網羅率を向上するのに役に立つことが考えられる。実際に実験後のアンケートの記述欄には、推薦コードによって見落としていたテスト項目をフォローすることができたという報告が複数存在した。

条件分岐が多く複雑なプログラムのテストコードを作成する際、SuiteRec の利用はカバレッジ (C1) を向上するのに役立つ可能性がある。

**RQ2:** SuiteRec はテストコードの作成時間を削減できるか？ RQ2 では、SuiteRec を使用した場合とそうでない場合で被験者のテストコード作成タスクが完了するまでに費やした時間を比較する。図 17 の結果を見ると、3 つのタスクの内 2 つのタスクで SuiteRec を使用した場合は、そうでない場合と比べてテスト作成時間が

<sup>6</sup><https://www.eclipse.org/>

<sup>7</sup><https://www.eclemma.org/>



図 15 命令網羅の平均カバレッジ

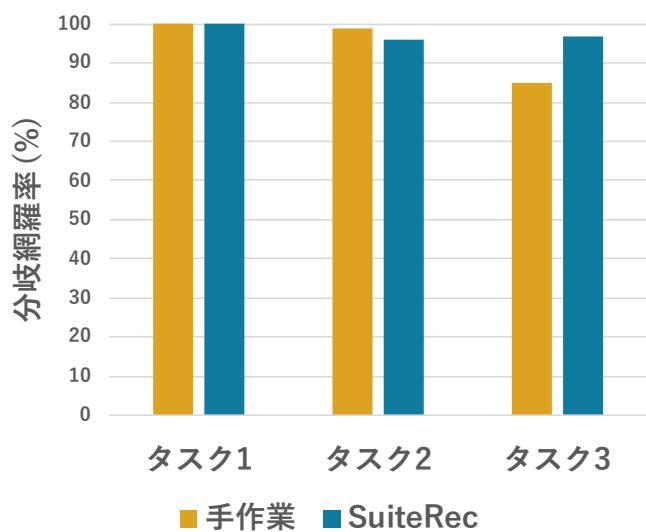


図 16 分岐網羅の平均カバレッジ

大きくなっていることが分かる。この結果は SuiteRec によって推薦される複数のテストスイートを読み理解するのに時間がかかる可能性がある。被験者はほとんどの場合、推薦されるテストコードをそのままの形で再利用することができない。入力したコード片と検出された類似コード片の差分を確認し、テストコードを書き換える必要がある。一方で、タスク 2 については、SuiteRec を利用した場合の方がテスト作成時間が短いことが分かる。我々は、提出されたテストコード調査したところ、カバレッジに差はないものの SuiteRec を使用しない場合は、テストケースの重複が多くなっていることが分かった。この結果は、被験者は無駄なテストケースを多く記述するのに時間を費やした可能性がある。

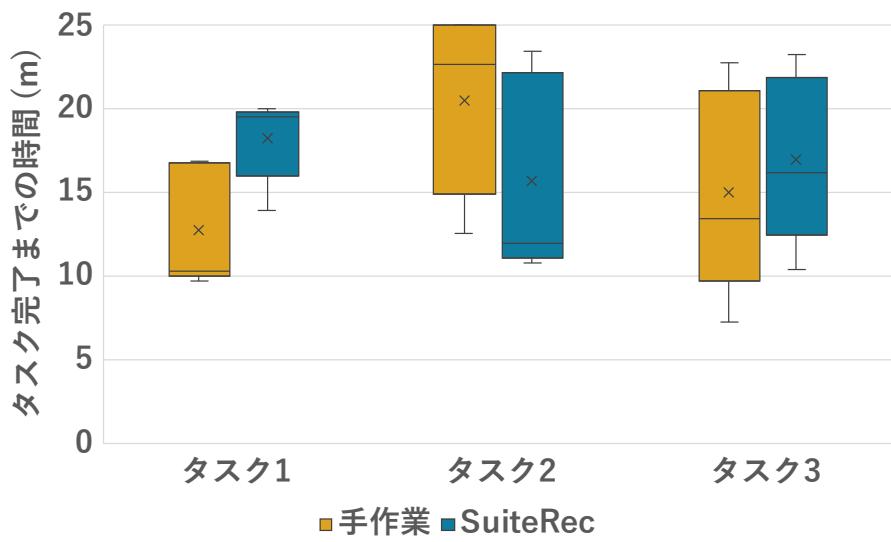


図 17 テストコード作成タスク終了までの時間

SuiteRec の利用は、推薦されたテストスイートを理解し変更する必要があるので、開発者はテストコード作成に多くの時間を費やす可能性がある。

**RQ3:** SuiteRec は、テストスメルの数が少ないテストコードの作成を支援できるか？ RQ3 では、SuiteRec を使用した場合とそうでない場合で被験者が作成したテストコード内に含まれるテストスメルの数を比較する。図 18 の結果は、すべての被験者が提出したテストコード内に含まれていたテストスメルの数の合計を示

す。すべてのタスクに対して、SuiteRec を使用して作成されたテストコードは、テストスメルをあまり含んでいないことが分かる。この結果は、SuiteRec によって推薦されるテストコード自体の品質が高く、被験者はそれを再利用することで品質を維持したままテストコードを作成できたと考えられる。また、SuiteRec の出力画面で推薦されるテストスイート内に含まれているテストスメルを提示することで、それを基にテストコードを書き替えることができ、品質が高いテストコードを提出した可能性が考えられる。一方で、SuiteRec を使用しなかった場合は、使用した場合と比べ全体として 5 倍以上、被験者はテストスメルを埋め込んでいた。その中でも多く埋め込まれていたテストスメルとして、“Assertion Roulette”, “Default Test”, “Eager Test” が挙げられる。多くの被験者は、初期状態のテストメソッドの名前を変更せず、一つのテストメソッド内でコピーアンドペーストによって assert 文を記述していたのが原因だと考えられる。実際に、既存研究でもこれらのテストスメルが、既存プロジェクトで多く検出されていることが報告されている [24]。

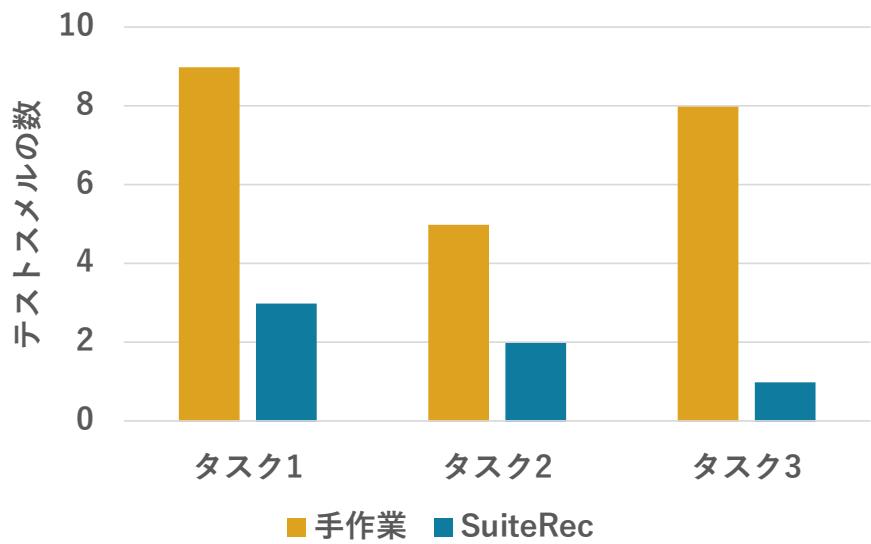


図 18 テストコード内に含まれていたテストスメルの数

開発者は、SuiteRec によって推薦される高品質のテストスイートを参考にすることで品質の高いテストコードを作成できる可能性がある。

**RQ4:** SuiteRec の利用は、開発者のテストコード作成タスクの認識にどう影響するか？ RQ4 では、評価実験の後、被験者に対して実験タスクに関するアンケートを実施した。図 19 は実施したアンケートの内容とその結果を示す。初めの 2 つの質問から、被験者は、実験タスクを明確に理解し（質問 1）、実験タスクを終えるのに十分な時間があったことが分かる（質問 2）。残りの質問については、SuiteRec を使用した場合とそうでない場合で、実験タスクに対する意見に違いがある。

質問 3 の回答から被験者は、テストコードを作成する際に、SuiteRec を用いるとテストコード作成を容易に感じることができる。しかし、この結果は実際のタスクの完了までの時間（図 17）とは対照的であり、SuiteRec を使用した場合の方がタスクの完了までにかかる時間が長いことが分かる。被験者は、推薦された複数のテストスイートを読み理解して再利用するかどうかを決定する。また、テストコードはそのままの状態で適用することはできず、入力コードと検出された類似コードの差分を理解し、テストコードに適切な修正を加える必要がある。我々は、SuiteRec を使用した場合、被験者はこの部分に多くの時間を費やすことがあると推測している。しかし、これらの作業は単純で繰り返すことが多いので被験者は容易に感じた可能性がある。また、推薦されたテストコードがテスト項目を考える上で参考になり、テストコードの記述には時間がかかるが全体としては容易な作業だと感じた可能性が高い。

質問 4 の回答から被験者は、SuiteRec を使用した場合、自身で作成したテストコードのカバレッジに自信があることが分かる。一方で、何も使用しなかった場合 40% の被験者がネガティブな回答を報告した。しかし、実際に提出されたテストコードのカバレッジには、ほとんど差がないことが分かる（図 15, 16）。開発者が自分自身で作成したテストコードのカバレッジに自信を持つことは重要である。開発者は、自分の書いたコードに責任を持ち、不安なくソフトウェアをユーザに提供できることは、ソフトウェアテストを行う目的の一つである。

質問 5 の回答から、何も使わずテストコードを作成した場合、40% の被験者が自

身の書いたテストコードの品質に自信が持てないことが分かった。実際に、提出されたテストコード内のテストスメルの数も SuiteRec を使用した場合よりも多く存在していることが分かる(図 18)。開発者は無意識の内にテストスメルを埋め込み、それが後の保守活動を困難にする。SuiteRec の利用は、開発者にテストコードの品質に対する意識を与えることでテストメルの数を減らし、作成したコードに自信をもたらす。一方で、SuiteRec を利用した場合でも品質に関してネガティブな意見も存在した。アンケートの記述項目では、テストスメルの存在は意識できたが具体的にどう修正して無くすことができるのか分からなかったと報告されていた。これは SuiteRec の更なる改善の必要性を示しており、各テストスメルに対するリファクタリング方法も提示する機能を追加すべきだと考えている。

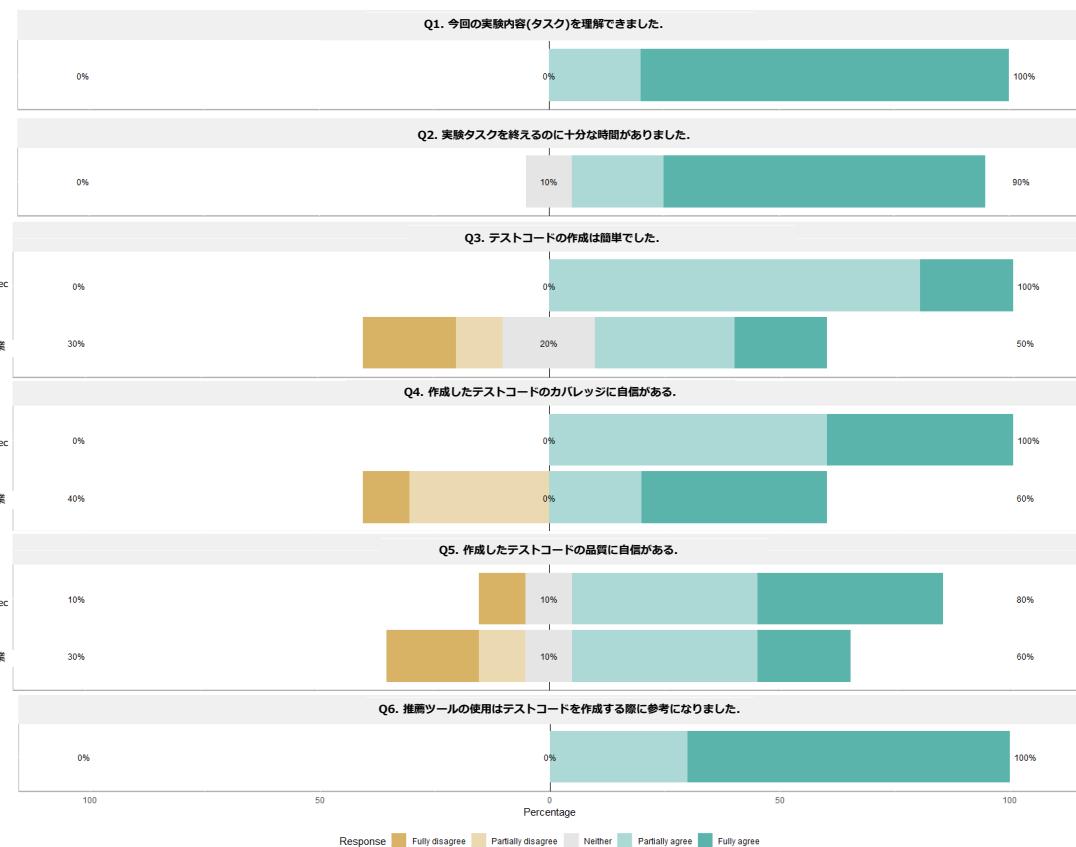


図 19 実験後のアンケートの回答

SuiteRec を利用した場合、開発者はテスト作成タスクを容易だと認識し、作成したテストコードに自信が持てる。

## 4.2 評価実験 2: 推薦されるテストスイートの順位付けに関する実験

評価実験 2 では、SuiteRec によって推薦される複数のテストスイートの順位付けの妥当性について検証する。我々は、被験者の数を増やすためにオンライン上で行うアンケートを実施した。被験者は、SuiteRec によって 1 位から 10 位まで順位付けられたテストスイートを読み、推薦されたテストスイートを参考にしたいかどうかを選択する。被験者のアンケート結果を基に、以下のリサーチクエスチョンに答えることを目指す。

**RQ5: SuiteRec は開発者が参考したいテストスイートを上位に推薦できるか？**

アンケート調査で、被験者が参考にしたいと回答したテストスイートが実際の SuiteRec の出力結果で上位に推薦されているかを評価する。評価方法では、検索エンジンなどの評価で用いられる代表的な評価指標を利用した。

以降、4.2.1 節では、評価実験 2 で用いた実験用データセットについて説明する。4.2.2 節では、評価実験 1 の手順について説明する。最後に 4.2.3 節で、評価実験 2 の結果について説明する。

### 4.2.1 評価実験 2 のデータセット

評価実験 2 を実施するために、SuiteRec によって推薦されるテストスイートの順位付けに関するアンケート調査を実施した。我々は被験者の数を増やすためにオンライン上で行えるアンケートを作成した。その結果、学生 11 人と社会人 3 人の計 14 人の被験者がアンケートに回答した。アンケートのパーソナリティに関する項目によると、9 割以上の被験者が 2 年以上のプログラミング経験があり、6 割以上の被験者が 2 年以上の Java の経験があった。また、すべての被験者が授業などの講義でソフトウェアテストに関する基本的な知識を持っていた。

評価実験2では、被験者に2つのタスクを割り当てた。以降、これらの2つのタスクをタスクA、タスクBと呼ぶ。被験者は、各タスクで与えられる関数単位のプロダクションコードのテストコードを作成する際に、SuiteRecによって推薦されるテストスイートが参考になるかどうかを選択する。以下に、各タスクの概要を示す。

### タスクA

タスクAは、会計処理を行うプログラムであり、ユーザが購入したアイテムの合計金額を計算するJavaメソッドである。以下に、タスクAのプロダクションコードの仕様とソースコードを示す。

#### タスクAのプロダクションコードの仕様

- 入力配列 item[ ] の要素の合計値を返す。
- 入力配列の要素数が0の時、-1を返す。
- 引数：整数型の配列 item[ ]
- 返り値：配列要素の合計値 totalprice

```
public int countPrice(int item[]){
    int totalprice = 0;
    if (item.length == 0){
        return -1;
    }
    for( int i=0; i < item.length; i++ ) {
        totalprice += item[i];
    }
    return totalprice;
}
```

図 20 タスクAのプロダクションコード

## タスク B

タスク B は、10 進数を 2 進数または、16 進数に変換するプログラムである。第 1 引数に応じて、第 2 引数の数値の変換方法を変更し結果を出力する Java メソッドである。以下に、タスク B のプロダクションコードの仕様とソースコードを示す。

### タスク B のプロダクションコードの仕様

- 第 1 引数が “bin” の場合、第 2 引数(10 進数)を 2 進数に変換する。
- 第 1 引数が “hex” の場合、第 2 引数(10 進数)を 16 進数に変換する。
- 第 1 引数が上記以外の場合、“can not conversion” を返す。

```
public String ConversionNum(String select,int dec) {  
    if (select == "bin") {  
        String num = Integer.toBinaryString(dec);  
        return num;  
    }else if(select == "hex") {  
        String num = Integer.toHexString(dec);  
        return num;  
    }else {  
        return "can not conversion";  
    }  
}
```

図 21 タスク B のプロダクションコード

### 4.2.2 評価実験 2 の手順

5.1 節で説明した、実験タスクを用いた評価実験 2 の手順について説明する。評価実験 2 は、Google Form を用いたオンライン上のアンケートで実施された。アンケートでは、最初に被験者間の事前知識の違いによる結果の相違をなくすために、ソフトウェアテストに関する基本的な資料と良いテストコードの設計に関する資料を提供した。また、アンケートの回答をする際には事前資料を参考にしな

がら回答することを推奨した。その後、被験者に2つの実験タスクから推薦される合計20個のテストスイートについて参考にしたいかどうかを選択してもらった。また、被験者には、なぜその回答を選択したのか、その理由を代表的な悪いテストコードの設計の選択肢から回答してもらった(自由回答)。最後に、実験タスク終了後に、被験者のパーソナリティに関するアンケートに回答してもらった。

#### 4.2.3 評価実験2の結果

本節では、SuiteRecによって推薦されるテストスイートの順位付けに関する評価実験の結果を報告する。そして、4.2節で説明したリサーチクエスチョン5について、実験結果を基に回答する。

**RQ5:** SuiteRecは、開発者が参考にしたいテストスイートを上位に推薦できるか？ RQ5では、15人の被験者によるアンケート結果を代表的な3つのランキングの評価指標に基づいて計算する。以下に、評価実験2で利用した評価指標について説明する。

#### Mean Reciprocal Rank (MRR)

MRRは、推薦リストを上位から確認し、最初に適合した要素の順位を計算に利用したランキング指標である。MRRは、0から1の値をとり、すべてのユーザに対して推薦リストの第1番目のアイテムが適合アイテムならば、1になる。一方で、正解が一つもリストに含まれない場合は、0になる。この指標の特徴として、ランキング上位での順位の差異は指標に大きく影響するが、下位での順位の差異はあまり影響を与えない。

$$MRR = \frac{1}{|U|} \sum_{u \in U} \frac{1}{k_u}$$

- $u$  は各対象ユーザ、 $U$  は全対象ユーザ
- $k_u$  はユーザ  $u$  への推薦リストのうち、最初に  $u$  が適合するアイテムが出現した順位

## Mean Average Precision (MAP)

MAP は、 AP(Average Precision) の値を対象のユーザ数で平均した値である。AP とは、適合アイテムが出現した時点をそれぞれ閾値として、閾値ごとの Precision を算出し、Precision を平均した値である。以下に AP と MAP の式を示す。

$$AP(u) = \sum_{k=1}^N \frac{Precision@k \cdot y_k}{\sum_{i=1}^k y_i}$$

$$y_k = \begin{cases} 1 : \text{上位 } k \text{ 番目が適合アイテム} \\ 0 : \text{それ以外} \end{cases}$$

$$MAP = \frac{1}{|U|} \sum_{u \in U} AP(u)$$

- $u$  は各対象ユーザ,  $U$  は全対象ユーザ

## Mean Recall@N (MR@N)

MR@N は、推薦ランキングの上位 N 番目の Recall を平均した値である。Recall@N とは、ユーザが実際に適合した上位 N 番目までのアイテムのうち、推薦リストでどれだけカバーできたかの割合である。以下に上位 k 番目の Recall@N と MR@N の式を示す。

$$Recall@N = \frac{|a \cap p_N|}{|a|}$$

- $N$ : 考慮する上位ランキングの数
- $a$ : ユーザが適合したアイテム集合
- $p_N$ : TopN の推薦リスト

$$MR@N = \frac{1}{|U|} \sum_{u \in U} Recall@N(u)$$

- $u$  は各対象ユーザ,  $U$  は全対象ユーザ

表 3 MAP と MRR の評価結果

	MAP	MRR
タスク A	84.1%	83.3%
タスク B	98.3%	100%
合計	91.2%	94.1%

表 4 MP@N の評価結果

Precision-topN	top1	top3	top5	top10
タスク A	73.3%	82.2%	85.3%	45.3%
タスク B	100%	97.8%	93.3%	53.3%
合計	86.7%	90.0%	89.3%	49.3%

上記の評価指標を基に計算した結果を表 3, 4 に示す.

表 3 は, MAP と MRR の評価結果を示す. 実験の対象とした 2 つのタスクに対して MAP は 80% を超えており, 高精度で開発者が参考にしたいテストスイートをランキングの上位に推薦できることが分かった. また, MRR についても各タスクに対して 80% の超えている. 一般的に, 開発者はランキングの上位に表示されるテストスイートを参考にしやすいことを考えると, 上位の含まれる適合アイテムの数が大きく影響する MRR の結果が高いことは, 重要である. 表 4 は, MP@K の評価結果を示す. 各タスクにおいて, top3 の精度は 80% を超えており SuiteRec は, ランキングの上位 3 位までに開発者が参考にしたいテストスイートを高精度で出力することができた.

各タスク毎の結果を比較すると, MAP, MRR, MP@N の 3 つの結果についてタスク A はタスク B よりも低いことが分かる. これはタスク A について SuiteRec が上位に推薦するテストコード内に含まれるテストスメルが関係していると考えられる. すなわち, SuiteRec のランキング手法は類似度を優先するためテストス

メルを多く含むテストスイートであっても、そのテストスイートに対応する類似コード片の類似度が高い場合、テストスイートは上位に推薦される。今回、タスク A について SuiteRec が上位に推薦したテストスイートは、類似度は高かったがテストスメルを含んでいた。従って、被験者の中には、上位で推薦されたテストスイートを参考にしたいと思わない被験者も存在した。一方で、タスク B については MRR と MP@N の結果から分かるように、SuiteRec によって上位に推薦されたテストスイートがテストスメルが少なく、類似度が高いことが分かる。このような場合、SuiteRec は高い精度で開発者が参考にしたいテストスイートを上位に推薦できる。今後は、SuiteRec のランキングについて、再利用しやすい類似度を優先とした並び替えと品質を重視したテストスメルの数を優先とした並び替えを開発者好みによって選択できるように、ランキング手法を拡張する必要がある。

SuiteRec は、開発者が参考にしたいテストスイートを高精度でランキングの上位に推薦できる。

## 5. 議論

本章では、4章の評価実験の結果から SuiteRec の有用性及び妥当性への脅威について議論する。

### 5.1 SuiteRec の有用性

本節では、4章の評価実験の結果を基に SuiteRec の有用性について考察する。RQ1では、作成したテストコードのカバレッジを測定した。被験者が実施したテストコード作成タスクにおいて、比較的単純な構造のプロダクションコード(タスク1, タスク2)のテストを作成する場合、SuiteRecの利用の有無でカバレッジ(C0, C1)にほとんど差がないことは重要な実験結果である。SuiteRecを使用した場合、テストコード作成に多くの時間を費やす可能性がある(RQ2)ことを考慮すれば、何も使用せずにテストコードを作成した方が開発時間を節約することができる。一方で、分岐数が多く複雑なプログラムのテストを作成する際、SuiteRecの利用はカバレッジ(C1)を10%以上向上できることが分かった。この結果は、推薦されたテストスイートは開発者がテスト項目を考える上で有益な情報になったと考えられる。実際に、実験後のアンケートの回答はこの考察を裏付けている。SuiteRecを利用した場合、すべての開発者は自分が作成したテストコードのカバレッジに自信があると回答した(質問4)。また、アンケートの自由記述欄の回答では、ある被験者は「テスト項目を推薦されたテストコードを見て、見落としていたテスト項目に気づくことができた」と回答した。これらの結果の有意性を検討するには被験者を増やした更なる調査が必要であり、開発者のテストコード作成スキルや経験に依存すると考えられる。

RQ3は、SuiteRecを使用して作成したテストコードはテストスメルの数が少なく品質が高いことを示した。開発者によって無意識に埋め込まれるテストスメルの存在は、後の保守活動を困難にする。SuiteRecはテストスメルの存在を被験者に意識させることで、保守性の高いテストコードの作成を支援できる可能性がある。実際にアンケートの自由記述欄の回答には「提示されたテストスメルを理解し、それをなくすようにリファクタリングしテストコードを作成した」という報

告がされている。さらに、「メソッド名を考える際に推薦されたテストコードのテストメソッド名が参考になった」という可読性向上に有益だったという回答も存在した。一方で、回答の中には「テストスメルの存在は意識できたが具体的にどう修正して無くすことができるのか分からなかった」という回答も存在した。これは SuiteRec の更なる改善の必要性を示しており、各テストスメルに対するリファクタリング方法も提示すべきだと考えている。

RQ5 は、SuiteRec は開発者が参考にしたいテストスイートを出力結果の上位に推薦できることを示した。SuiteRec のランキング手法では、類似度とテストスメルの数の 2 つの要素に基づいて順位付けされており、再利用しやすく品質の高いテストコードをランキングの上位に提示する。RQ3 の結果を考慮すると SuiteRec の出力結果の上位に推薦されるテストスイートを参考にすることで、開発者は高品質のテストコードを作成できる可能性がある。

## 5.2 妥当性への脅威

本節では、提案ツール SuiteRec と評価実験における妥当性の脅威を説明する。

SuiteRec は、入力コード片に対応する類似コード片を OSS プロジェクトから検索し、その類似コード片に対応するテストスイートを推薦する。そのため、入力コード片の類似コード片が見つからないとテストスイートを推薦できない。また、類似コード片を検出できたとしても類似度が低い場合、推薦されるテストスイートは参考にならない可能性がある。Nicad が検出するコードクローンはタイプ 2, 3 でありタイプ 3 のような命令文の挿入・削除が複数存在するような類似コード片は振る舞いが異なる場合がある。このような場合は、テストコードを再利用することは困難であり、推薦できるテストスイートは狭い範囲に限定される可能性がある。

SuiteRec で利用した Nicad は、検索対象のプロジェクトに対して一度に検索できるサイズに制限がある。従って、検索対象が大規模なプロジェクトになると計算コストが高い。しかし、Nicad の代わりに SourcererCC[29] や FaCoY[15] などのコード検索エンジンを使用すれば、大規模なプロジェクトに対しても一度の処理で検索することが可能である。本研究では、テストコードの再利用性を考慮し類

似度の高いコード片を検出できる Nicad を使用した。また、検出時間を短縮化するため検索処理を並列化するように SuiteRec に実装した。今後は、SuiteRec を他のコードクローン検出ツールにも対応させ、テストコードを推薦するまでの時間および推薦されたテストコードの違いを確認する必要がある。

SuiteRec の評価では、被験者による実験を行い SuiteRec を利用した場合とそうでない場合で作成されたテストコードを比較することで、有用性を示している。そのため、被験者のプログラミングスキルやテストコードの作成経験が実験結果に影響を与える可能性がある。ただし、アンケートのパーソナリティに関する項目によると 9 割の被験者が 2 年以上のプログラミング経験があり、その内 8 割の被験者が単体テストの作成経験があると回答した。また、実験前にソフトウェアテストに関する講義と練習問題を実施しており、被験者による本実験への影響は少ないと考えられる。一方で、評価実験 1 では 3 つの実験タスクを使用したが、今回の結果が使用された実験タスクに限定される可能性がある。評価実験 1 では、被験者がテスト対象のプロダクションコードの仕様を十分理解していることが前提となるので、競技プログラミングで用いられるコードを利用した。しかし、これが実際のソフトウェアプロジェクトで使用されるプロダクションコードの場合結果が変わる可能性がある。今後、OSS プロジェクトの実用コード片に対して実験を行い、結果がどのように変化するのか確認する必要がある。

評価実験 2 では、被験者を多くするためにオンライン上のアンケートで実験を実施した。評価実験 2 を実施した 15 人の被験者の内、9 人は評価実験 1 も実施している。そのため、評価実験 1 の背景知識が評価実験 2 に影響した可能性がある。今後は、評価実験 1 を実施していない被験者の数をさらに増加させ一般性を示す必要がある。

## 6. 関連研究

本章では、提案ツール SuiteRec の基本となるアイディアであるクローンペア間でのテストコード再利用に関する既存研究を紹介する。

Zhang ら [33] はクローンペア間でコードを移植を行い、移植前と移植後のテスト結果を比較しその情報を基に既存テストコードを再利用するツール Graftor を提案した。有名 OSS プロジェクトを対象とした Graftor の評価実験では、コードクローン検出ツール DECKARD[13] によって検出されたどちらか片方のコード片 テストコードが存在するクローンペア 52 個の内 94% でコード移植に成功し、テストコードを再利用できることを示した。

Soha ら [18] は、コード片を再利用する時にそのコード片に対応するテストスイートの関連部分を半自動で再利用および変換を行うツール Skipper を提案した。Skipper のアプローチは、再利用元のコード片のテストスイートを動的解析し、どのテストケースが再利用先のコード片を実行するかを特定する。そして、コード片の再利用タスクを支援するツール Gilligan[11, 12] に基づいて、テストスイートを変換し再利用する。Skipper の変換プロセスでは、開発者が変換プロセスを導くための詳細な再利用計画を決める必要がある。被験者による Skipper の評価実験では、被験者の手作業でのテスト作成タスクと比較して、テスト作成時間の短縮化できカバレッジが向上したことを示した。

SuiteRec はこれらのツールとは、2つの視点で異なる。1つ目は、SuiteRec は OSS 上のリポジトリから類似コード検索する。大規模なソースコードリポジトリ内で検索をかけることで、多くのテストスイートを見つけることができる可能性がある。次に、SuiteRec はテストスイートを推薦するだけであり、クローンペア間の詳細なテスト再利用計画は開発者に委ねていること。たとえ、自動的にテストを再利用できたとしても品質の低いコードを拡散することは、後の保守活動を困難にさせる。テストスメルを提示し、開発者自身にリファクタリングさせることで作成したテストに自信が持てると考える。

## 7. まとめと今後の課題

本研究では、コードクローン検出技術を用いたテストコード自動推薦ツール SuiteRec を提案した。SuiteRec は、開発者から入力された関数単位のコード片に対して類似コード片を検出し、その類似コード片に対応するテストスイートを推薦する。さらに、テストコードの良くない実装を表す指標であるテストスメルを開発者に提示し、より品質の高いテストスイートを推薦できるように推薦順位がランキングされる。SuiteRec の評価では、被験者によって SuiteRec の使用した場合とそうでない場合でテストコードの作成してもらい、テスト作成をどの程度支援できるかを定量的および定性的に評価した。その結果、SuiteRec の利用は条件分岐が多く複雑なプログラムのテストコードを作成する際にコードカバレッジの向上に効果的であること、作成したテストコードの内のテストスメルの数が少なく品質が高いことが分かった。また、SuiteRec は開発者が参考にしたテストスイートを上位に推薦できることを確認した。実験後のアンケートによる定性的な評価では、SuiteRec を使用した場合被験者はテストコードの作成が容易になると認識し、また自分の作成したコードに自信が持てることが分かった。

今後の課題として、以下が挙げられる。

- より実践的な利用に備えてツールを改善する必要がある。具体的には、SuiteRec に推薦されるテストコードを自動編集できる機能とテストスメルに対するリファクタリング方法を提示する機能を追加する予定である。
- SuiteRec の有意差を測るために評価実験 1 では、被験者を増やした更なる実験が必要である。また、経験値による効果の影響をなくすために対象の被験者を学生だけでなくテストコード作成経験の豊富なエンジニアに対しても参加してもらう必要がある。
- SuiteRec で採用したコードクローン検出ツール NiCad だけでは、検出できるコードクローンの数に限りがある。検出できるコードクローンの幅を広げるために SuiteRec を複数のコードクローン検出ツールに対応できるようにし、より多くテストスイートを推薦できるようにする必要がある。

## 謝辞

本研究を進めるにあたり、多くの方々に御指導及び御助言頂きました。ここに謝意を添えて御名前を記させて頂きます。

奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 飯田 元 教授には、ご多忙にも関わらず、毎週の進捗報告ミーティングや論文輪講に御参加して頂き、その中で常に本質的な御指導及び御助言を頂きました。また、本研究に限らず、GEIOT や enPiT など学外のプログラムに参加させていただき 2 年間充実した大学院生活を送ることができたことは、先生の御指導と御人柄によるものと確信しております。心より深く感謝を申し上げます。

奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 市川 昊平 准教授には、毎週の進捗報告ミーティングや論文輪講を通して、本研究に関して常に適切な御指導及び御助言を頂きました。また、本研究に限らず、enPiT プログラムの参加やタイ留学など大変貴重な経験をさせて頂き、充実した大学院生活を送ることができました。心より深く感謝を申し上げます。

京都工芸纖維大学情報工学課程 崔恩灝 助教には、本研究を進めるにあたって、積極的に打ち合わせや連絡を取って頂き、その中で終始適切な御指導及び御助言を頂きました。さらに、京都工芸纖維大学に移転した後も論文投稿時や論文輪講に関して常に適切な御助言をして頂き、研究室に配属されてから 2 年間、大変お世話になりました。また、本研究に限らず就職活動などでも助言を頂いたことや積極的に学生に関わって頂き、充実した生活を送ることができたことは、先生の御指導と御人柄によるものと確信しております。心より深く感謝を申し上げます。

奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 高橋 慧智 助教には、本研究の方向性や問題点などに関して、多くのご意見を頂きました。また、本研究についてだけではなく、専門知識に関して様々なご教授を頂き、ソフトウェア工学に止まらず多くのことを学ばさせて頂きました。本論文ならびに研究生活において、常に多くのご意見を賜りました心から御礼申し上げます。

最後に、その他様々な御指導及び御助言等を頂いた奈良先端科学技術大学院大学先端科学技術研究科情報科学領域ソフトウェア設計学研究室・超高信頼ソフトウェアシステム検証学研究室の皆様に深く感謝いたします。

## 参考文献

- [1] S. Abdelilah, P. Gilles, and Y. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering (TSE)*, 41:294–313, 2015.
- [2] F. Appel. *Testing with Junit*. Packt Publishing, 2015.
- [3] B. S. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering (TSE)*, 33(9):608–621, 2007.
- [4] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.
- [5] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377, 1998.
- [6] K. L. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.
- [7] E. Daka, J. Campos, G. Fraser, J. Dorn, and w. Weimer. Modeling readability to improve unit tests. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 107–118, 2015.
- [8] A. Deursen, L. M. F. Moonen, A. Bergh, and G. Kok. Refactoring test code. Technical report, 2001.
- [9] M. Ellims, J. Bridges, and D. C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.
- [10] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

- [11] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 447–457, 2007.
- [12] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering (TSE)*, 21(4), 2013.
- [13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.
- [14] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 301–310, 2011.
- [15] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. Le Traon. Facoy - a code-to-code search engine. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 946–957, 2018.
- [16] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: Program-analysis-guided random testing (t). In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 212–223, Nov 2015.
- [17] P. Machado and A. Sampaio. *Automatic Test-Case Generation*, pages 59–103. Springer Berlin Heidelberg, 2010.
- [18] S. Makady and R. Walker. Validating pragmatic reuse tasks by leveraging existing test suites. *Software: Practice and Experience*, 43:1039–1070, 2013.
- [19] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

- [20] F. Palomba, D. D. Nucci, A. Panichella, R. Oliveto, and A. D. Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 5–14, 2016.
- [21] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. D. Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 130–141, 2016.
- [22] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.
- [23] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 547–558, 2016.
- [24] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the International Conference on Computer Science and Software Engineering (CASCON)*, pages 193–202, 2019.
- [25] I. S. W. B. Prasetya. T3: Benchmarking at third unit testing tool contest. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 44–47, 2015.
- [26] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55:1165–1199, 2013.

- [27] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.
- [28] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74:470–495, 2009.
- [29] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, page 1157–1168, 2016.
- [30] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018.
- [31] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12, 2018.
- [32] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, page 4–15, 2016.
- [33] T. Zhang and M. Kim. Automated transplantation and differential testing for clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 665–676, 2017.
- [34] 倉地 亮介, 崔 恩灝, 飯田 元. 類似コード検出ツールを用いたテストコード再

利用に向けた調査. 第 26 回ソフトウェア工学の基礎ワークショップ (*FOSE*),  
pages 177–178, 2019.

- [35] 丹野 治門, 倉林 利行, 張 曜晶, 伊山 宗吉, 安達 悠, 岩田真治, 切貫 弘之. テスト入力値生成技術の研究動向. *コンピュータ ソフトウェア*, 34(3):121–147, 2017.
- [36] 肥後芳樹, 楠本真二, 井上 克郎. コードクローン検出とその関連技術. *電子情報通信学会論文誌*, J91-D(6):1465–1481, 2008.

これはおまけの図です。

図 22 おまけの図

## 付録

A. おまけその 1

B. おまけその 2