

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[†] 崔 恩瀬^{††} 飯田 元[†]

† 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 〒630-0192 奈良県生駒市高山町 8916-5

†† 京都工芸繊維大学情報工学課程 〒606-8585 京都府京都市左京区松ヶ崎橋上町

E-mail: †kurachi.ryosuke.kp0@is.naist.jp, ††echoi@kit.ac.jp, †††iida@itc.naist.jp

あらまし テスト工程において、テスト作成コストを削減するために様々なテストコード自動生成ツールが提案されてきた。しかし、既存のツールによって生成される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いないという性質から開発者の保守作業を困難にする課題がある。この課題の解決方法として、本研究では OSS プロジェクト上に存在する既存の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。また、被験者実験を行い SuiteRec の有用性を確認した。

キーワード 類似コード検出, 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

SuiteRec: Automatic Test Suite Recommendation System based on Code Clone Detection

Ryosuke KURACHI[†], Eunjong CHOI^{††}, and Hajimu IIDA[†]

† Graduate School of Information Science, Nara Institute of Science and Technology 8916-5, Takayama, Ikoma, Nara, 630-0192, Japan

†† Department of Information Science Matsugasaki, Sakyo-ku, Kyoto, 606-8585 Japan

E-mail: †kurachi.ryosuke.kp0@is.naist.jp, ††echoi@kit.ac.jp, †††iida@itc.naist.jp

Abstract Automatically generated tests tend to be less read-able and maintainable since they often do not consider the latent objective of the target code. Reusing existing tests might help address this problem. To this end, we present SuiteRec, a system that recommends reusable test suites based on code clone detection. Given a java method, SuiteRec searches for its code clones from a code base collected from open-source projects, and then recommends test suites of the clones. It also provides the ranking of the recommended test suites computed based on the similarity between the input code and the cloned code. We evaluate SuiteRec with a human study of ten students. The results indicate that SuiteRec successfully recommends reusable test suites.

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

1. はじめに

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。これまでにテスト工程を支援するために、様々な自動生成技術が提案してきた。[?], [?], [?], [?], [?].

EvoSuite [?] 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 を用いて単体テストを自動生成することで、開発者は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、EvoSuite などの既存ツールによって自動生成されるテストコードは、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で、開発者の保守作業を困難にさせる [?], [?], [?]. 開発者は、テ

ストが失敗するたびにテスト対象コード内で不具合の原因を特定または、テスト自体を更新するか否かを判断する必要がある。Shamshiri ら [?] は、自動生成されたテストコードは可読性が低く、開発者がテスト対象コードの不具合を特定するのに、効果的でないことを報告した。

我々は、この課題を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を推薦するツール SuiteRec を提案する。推薦手法のアイディアは、類似するソースコード間でテストコード再利用することである。SuiteRec は、入力コード片に対する類似コード片を検出し、その類似コード片に対するテストスイート(テスト項目のまとま

り)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を並び替える。

SuiteRec の有用性を評価した被験者実験では、*SuiteRec* を使用した場合とそうでない場合で、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テストコードに自信が持てることが分かった。

以降、2章では、本研究に関わるテストスメルおよびテストコード自動生成技術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 テストスメル

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。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 [?] によって提唱された。さらに、Deursen ら [?] 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 [?] によって拡張された。

以下に、本研究で扱う6種類のテストスメルを説明する。

Assertion Roulette: テストメソッド内に複数の assert 文が存在するテストコード。各 assert 文は異なる条件をテストするが、テストが失敗した場合開発者へ各 assert 文のエラーメッセージは提供されないので、失敗を特定することが困難になる。

Conditional Test Logic: テストメソッド内に複数の制御文が含まれているテストスイート。テスト成功・失敗は制御フロー内にある assert 文に基づくの予測するのが難しい。

Default Test: JUnitなどのテスティングフレームワークを使用したテストコードの内、テストクラスやテストメソッドの名前がデフォルトの状態であるテストコード。テストコードの可読性の向上ために適切な名前に変更する必要がある。

Eager Test: テスト対象クラス内の複数のメソッドを呼び出しているテストコード。テストメソッド内で複数のメソッド呼び出しを行うと、何をテストしているかについて混乱が生じる。

Exception Handling: テストメソッド内で例外処理が含まれているまたは、例外を投げるテストコード。例外処理は対象コードに記述し、テストコード内で例外処理が正しく行われる

かどうかを確かめるようにリファクタリングする必要がある。

Mystery Guest: テストメソッド内で、外部リソースを利用するテストコード。テストメソッド内だけで完結せず外部のファイルなど、外部リソースを利用すると外部との依存関係が生じ、外部リソースが壊れた場合テストも失敗してしまう。

2.2 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。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。EvoSuite [?] 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。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かうことができる。しかし、EvoSuite などの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ので、開発者の保守作業を困難にするという課題がある [?], [?], [?]. 開発者は、テストが失敗するたびに、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。Shamshiri らは、自動生成されたテストコードは可読性が低く、開発者がテスト対象コードの不具合を特定するのに、効果的でないことを報告した。また、Palomba ら [20] は、EvoSuite によって自動生成されたテストコードは、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。

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。*SuiteRec* の概要を図1に示す。*SuiteRec* は、主に以下の4つのステップから構成される。

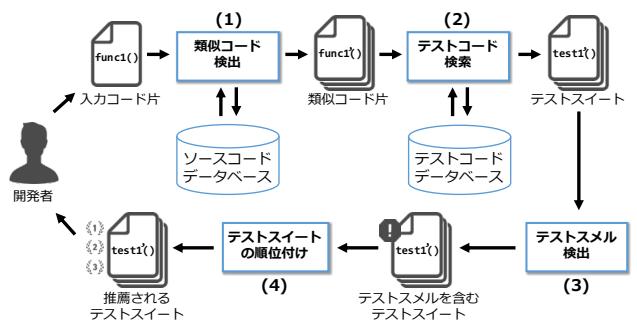


図1 *SuiteRec* の概要

Step1: 類似コード検出ツールを用いて、開発者から与えられた入力コード片に対する類似コード片を検出する。

Step2: Step1で検出された複数の類似コード片に対するテス

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: Step2 で検出された複数のテストスイート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を並び替える。

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、開発者から与えられ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 [?] 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、高精度・高再現率でクローンペアの検出するツールである。

SuiteRec は、NiCad を用いて入力コード片に対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ソースコードデータベースから検索する。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^(注1) 上の 3,205 個の OSS 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たデータセット [?] の内、テストフォルダが存在し、JUnit のテ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。そこで、本研究で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行い、ソースコードデータベースに格納した。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片の検索を実現した。

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するテストスイートを検索する。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）からテストスイートを検出するために、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、テスト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。

Phase1: 命名規則によるクラス単位で、テストクラスとテスト対象クラスを対応付ける。

Phase2: テストコードを静的解析し、各テストコードから呼び出されるすべてのテスト対象のメソッド名を収集する。

Phase3: テストメソッド名を区切り文字や大文字で分割し、テスト対象のメソッド名と部分一致した場合、テストコードとテスト対象コードをメソッド単位で対応付ける。

本研究は、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。Phase1 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、テスト対象クラスである “Calculator クラス” とテストクラスである “CalculatorTest クラス” が対応付けられる。

Phase2 では、テストコードを ANTLR^(注2) を用いて静的解析し、テストメソッド内で呼び出されるテスト対象のメソッド名を取得する。単体テストは、一般的に、図 2 の例のようにテストコード内でテスト対象のオブジェクトの生成を行い、テスト対象のメソッド呼び出して実行される。したがって、TCD 内のテストコードを静的解析し、テスト対象の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、複数のテスト対象のメソッドが呼び出されていることも考えられるので、Phase3 では、さらにテスト対象のメソッド名とテスト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、テストメソッド名にテスト対象メソッドの名前が記述されていることが多い [?]. したがって、テストメソッドの名前を区切り文字や大文字で分割し、テスト対象のメソッド名と部分一致した場合、テストコードとテスト対象コードをメソッド単位で対応付けるように実装した。

```
public class Calculator {  
    public int multiply(int x, int y) {  
        return x * y;  
    }  
}  
  
@Test  
public void testMultiplyOfTwoNumbers() {  
    Calculator calc = new Calculator();  
    int expected = 200;  
    int actual = calc.multiply(10, 20);  
    assertEquals(expected, actual);  
}
```

図 2 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に対して、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^(注3) [?] 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% の検出精度と 90%～100% の再現率でテストスメルを検出できる。tsDetect は、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、2.1 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。また、再利用対象の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、SuiteRec が推薦す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

(注2) : <https://www.antlr.org/>

(注3) : <https://testsmeells.github.io/>

- **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、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。

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[?]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。具体的には、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テストスイートを並び替え、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。

4. 評価実験

この章では、SuiteRec の有用性を定量的および定性的に評価するために行った評価実験について説明する。

評価実験では、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、SuiteRec がテストコードの作成をどの程度支援できるかを評価した。具体的には、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で、作成した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および、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 4 つ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？
- **RQ2:** SuiteRec はテストコードの作成時間を削減できるか？
- **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？
- **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？

4.1 評価実験のデータセット

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。被験者がテストコードを作成するためには、プロダクションコードの仕様を十分に理解しておく必要がある。そこで、本研究で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。図 3 は、各タスクの概要を示す。

	Task1	Task2	Task3
概要	典型的なFizzBuzz の関数	第1引数に応じて計算方法を変更し、計算結果を返す	2つの入力値に基づいて試験の合否を判定する
分岐数	8	16	24

図 3 実験タスクの概要

4.1.1 評価実験の手順

4.1 節で説明した実験タスクを用いた評価実験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 SuiteRec の利用方法とテストコードの作成について十分に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、実験タスクを終了を宣言してもらい、実験タスク開始から終了宣言までの時間をタスク完了までの時間とした。実験時間は 1 つのタスクにつき最大 25 分の時間を設け、それ以降はタスクの途中だとしても作業終了してもらった。

我々は、SuiteRec の利用効果がタスクによって偏らないように、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。

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。

4.1.2 評価実験の結果

本節では、評価実験の結果を、本章の前半で説明した 4 つのリサーチクエスチョンを用いて説明する。

- a) **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？

この RQ に答えるために、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のカバレッジを比較した。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^(注4) のプラグインであ

(注4) : <https://www.eclipse.org/>

る EclEmma^(注5) を利用した。図 4 と図 5 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。この図から分かるように、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 5 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、推薦されたテスト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。

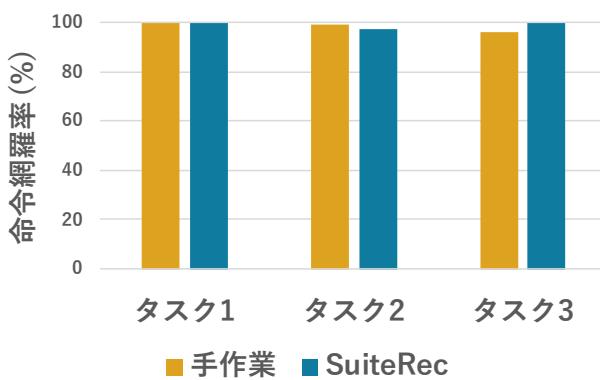


図 4 命令網羅の平均カバレッジ

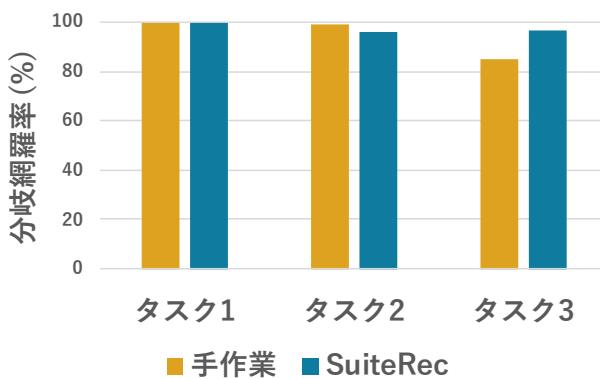


図 5 分岐網羅の平均カバレッジ

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。

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

この RQ を答えるために、SuiteRec を使用した場合とそうで

(注5) : <https://www.eclemma.org/>

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した。図 6 は、各被験者のタスク完了までに費やした時間の分布を示す。この図から分かるように、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長いことが分かる。SuiteRec を用いた場合、テスト作成に時間がかかる原因として、推薦される複数のテストスイートのソースコードを理解し、再利用する際に変更する必要があることが考えられる。被験者は多くの場合、推薦されるテストコードをそのまま再利用できない。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を調査したところ、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いことが分かった。この結果から、SuiteRec の利用は、被験者が無駄なテストケースを作成することを防ぎ、短時間でのテストコード作成を支援できることが分かった。

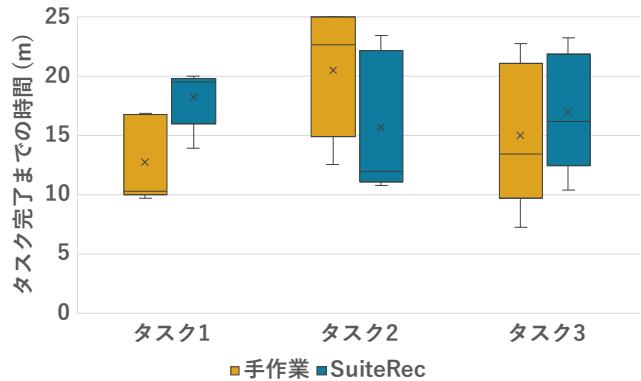


図 6 テストコード作成タスク終了までの時間

SuiteRec の利用は、開発者のテストコード作成に多くの時間を費やす。しかし、無駄なテストケースの作成するのを防ぎ、短時間でのテストコード作成を支援できる場合もある。

- c) 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？

この RQ に答えるために、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した。図 7 は、各タスクごと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。この図から分かるように、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スイートは、使用しない場合と比べて検出されたテストスメルの数が少ないことが分かる。これは、SuiteRec によって推薦されるテストスイートの品質が高く、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の情報を提示することで、その情報に基づいて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 を使用せずに作成されたテストスイートは、SuiteRec を使用して作成されたテストスイートと比べ全体として 5 倍以上のテストスメルが含まれていた。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。これは多くの被験者が、初期状態のテストメソッドの名前を変更せず、1 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` 文を記述していたこと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[?].

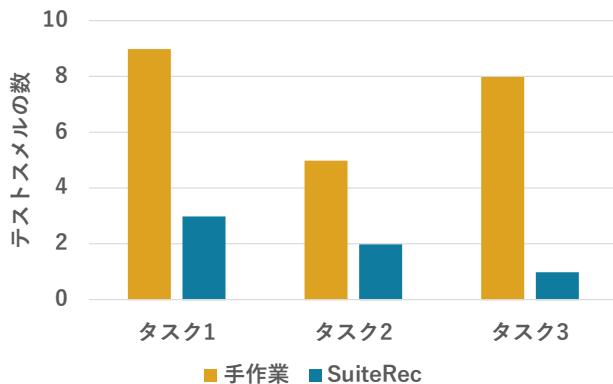


図 7 テストコード内に含まれていたテストスメルの数

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。

d) 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？

この RQ に答えるために、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 8 は実施したアンケートの内容とその結果を示す。Q1, Q2 の回答から、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)。Q1, Q2 以外の質問については、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 を用いることでテストコード作成を容易に感じることが分かった。しかし、この結果は実際のタスクの完了までの時間(図 6)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。SuiteRec を使用した場合、被験者はテストコードの作成に多くの時間を費やす。しかし、SuiteRec を用いたテストコードの作成作業は、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、SuiteRec によって推薦されたテストスイートがテスト項目を考える上で参考になり、テストスイート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、SuiteRec を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 4, 5)。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、自分の作成し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 1 つである。

質問 5 の回答から、SuiteRec を使用せずにテストコードを作成した場合、40% の被験者が自身の作成したテストコードの品質に自信が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(図 7)。開発者は無意識の内にテストスメルを埋め込み、そのテストスメル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スメルの数を減らし、作成したコードに対して自信をもたらす。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である。

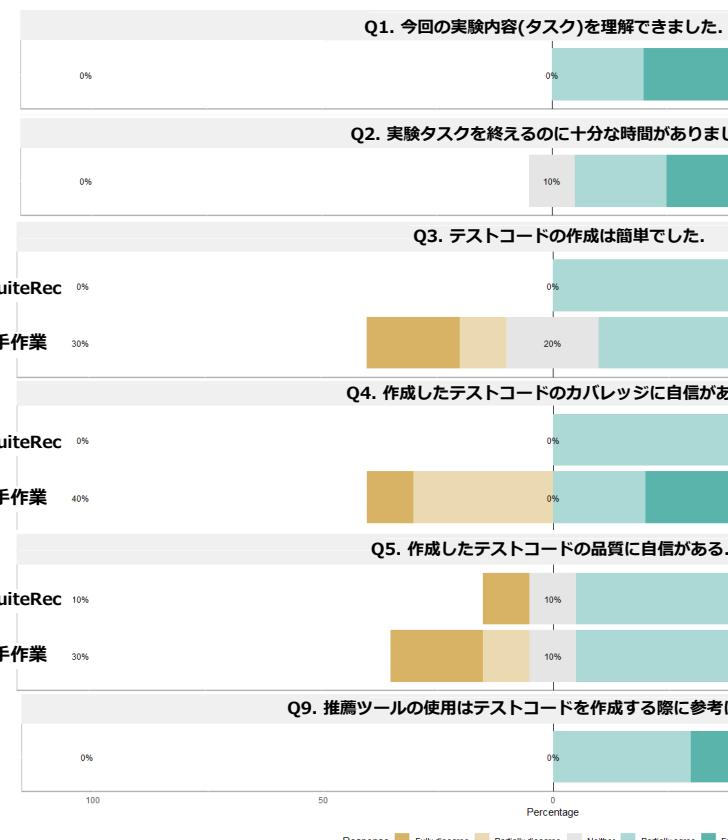


図 8 実験後のアンケートの回答

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。

5. 議論

6. まとめと今後の課題