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 恩瀞 准教授	（京都工芸繊維大学）

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されてきた．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後のメンテナンス活動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 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．SuiteRec は，類似コード検索ツールを用いて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に対して類似コードを検出し，その類似コード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を示す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．提案ツールの評価では，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) 開発者はテストの作成を容易だと認識し，自身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．

*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

類似コード検出, 推薦システム, ソフトウェアテスト, 単体テスト

Automatic Test Suite Recommendation System based on Code Clone Detection*

Ryosuke Kurachi

Abstract

Automatically generated tests tend to be less read-able and maintainable since they often do not consider thelatent objective of the target code. Reusing existing tests mighthelp address this problem. To this end, we present SuiteRec, asystem that recommends reusable test suites based on code clonedetection. Given a java method, SuiteRec searches for its codeclones from a code base collected from open-source projects,and then recommends test suites of the clones. It also providesthe ranking of the recommended test suites computed basedon 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 reusabletest suites.

Keywords:

clone detection, recommendation system, software testing, unit test

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

目 次

図 目 次

表 目 次

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 [1]．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 [?]．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 [?],[?],[?],[?],[?].

既存研究で提案されている 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，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 [?],[?],[?]．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提示させる．テストが失敗するたびに，開発者はテスト対象のプログラム内での不具合を原因を特定するまたは，テスト自体を更新する必要があるかどうかを判断する必要がある．自動生成されたテストは，自動生成によって得られる時間の節約よりも読みづらく，保守作業に助けになるというよりかむしろ邪魔するという結果が報告されている [?].

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 SuiteRec を提案する．推薦手法の基本となるアイディアは類似コード間でのテストコード再利用である．SuiteRec は，入力コードに対して類似コードを検出し，その類似コード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．

提案ツールの評価では，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，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述べる．3章では，本研究で提案するテストコード自動推薦手法について述べる．4章では，被験者による提案ツールの評価実験について述べる．5章では，提案ツールの評価実験の考察について述べる．最後に，6章でまとめと今後の課題について述べる．

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．[10]．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 [1,2,8] 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 [16,24] 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 [10,23]．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を意味する．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書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，テスト管理とうい大きく 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，結合テスト Selenium，Appium 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．

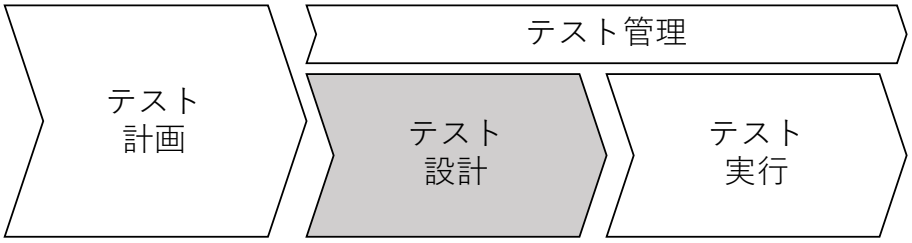

図 1 テストにおけるタスク

2.2.1 単体テスト

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．

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

2.2.2 テストコードの品質

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 [] によって提唱された．さらに，Van Deursen ら [50] 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，18 個の新しいテストスメルを定義した Meszaros [42] によってより拡張された．

この節では，本研究で扱う以下の 6 種類のテストスメルを紹介する．

- 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 文が存在する場合発生する．各 assert 文は異なる条件をテストするが，開発者へ各 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．

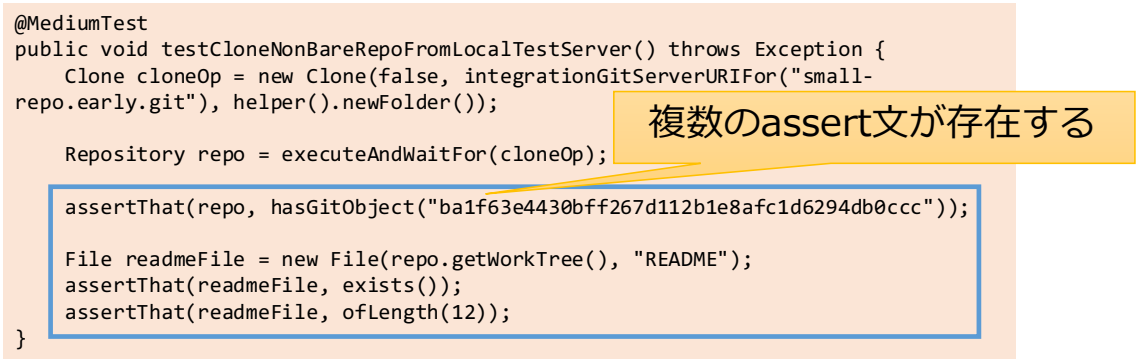

図 2　Assertion Roulette の例

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．

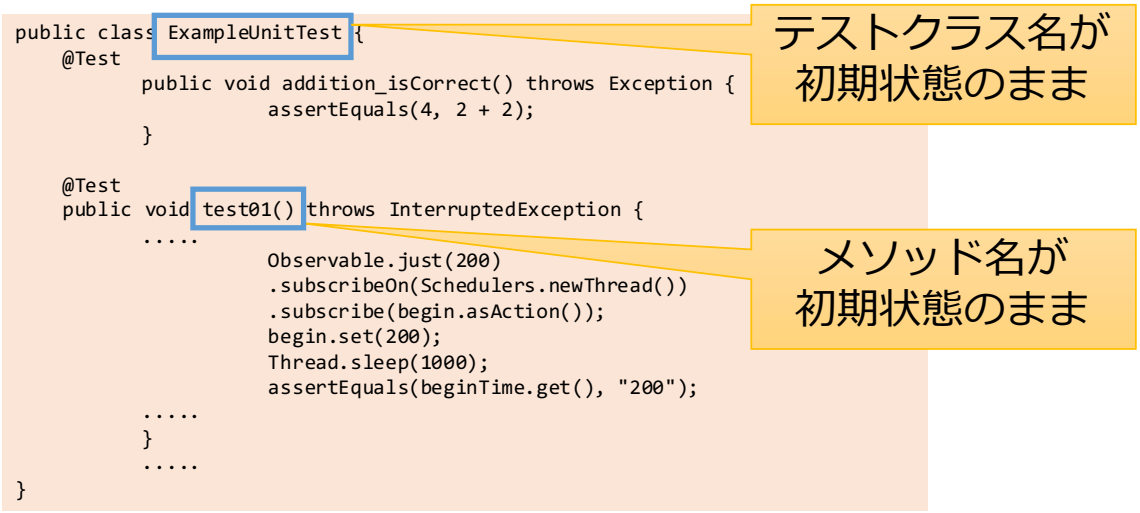

図 3 Default Test の例

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

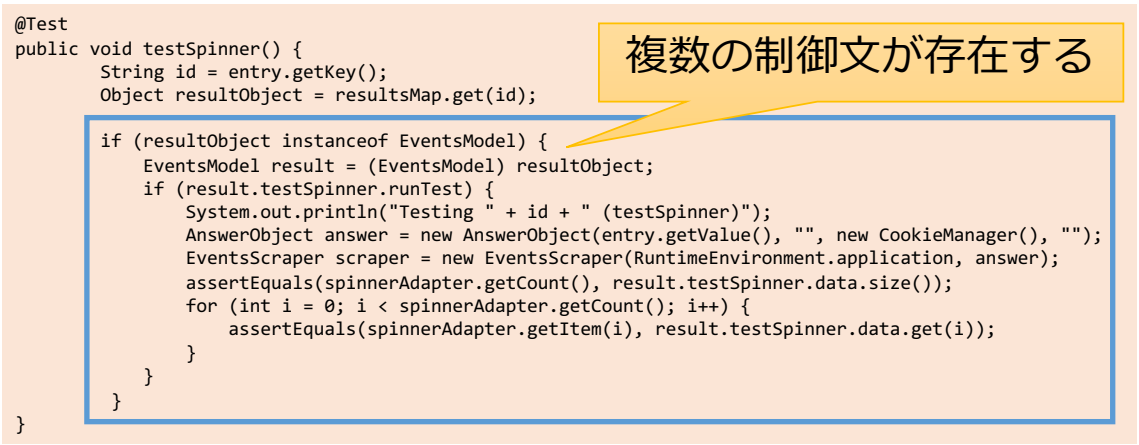

図 4 Conditional Test Logic の例

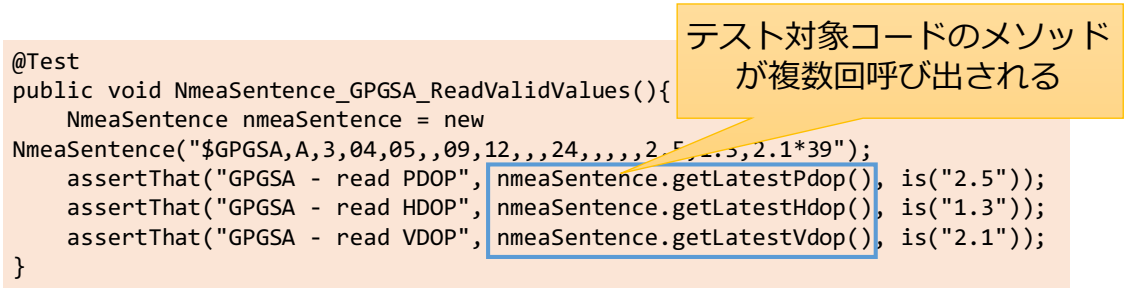

図 5 Eager Test の例

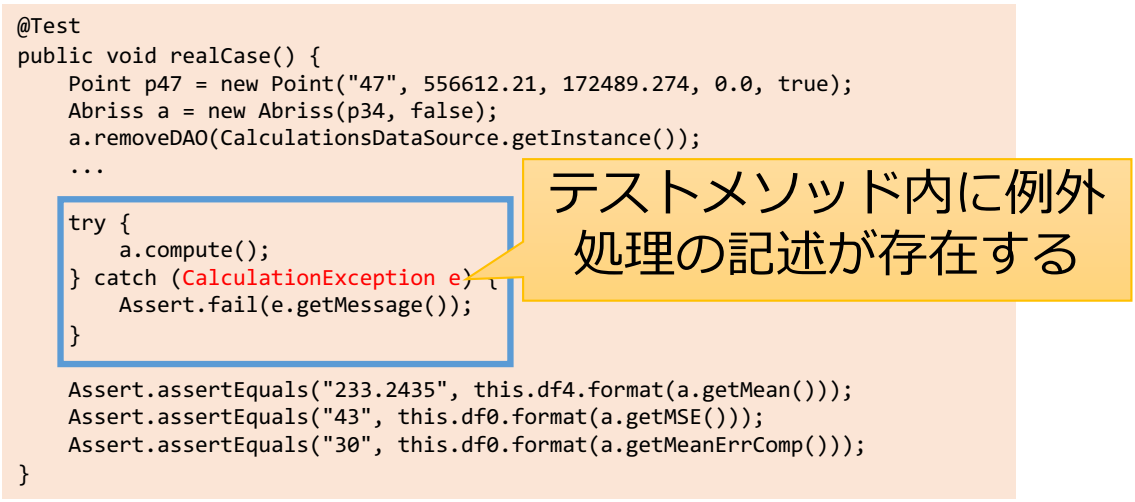

図 6 Exception Handling の例

た場合発生する．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．

.

Bavota ら [7] 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．調査結果は，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の存在が影響を受けるクラスの理解に悪影響を及ぼすことを明らかにした．

Tufano[71] らは，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 [] 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

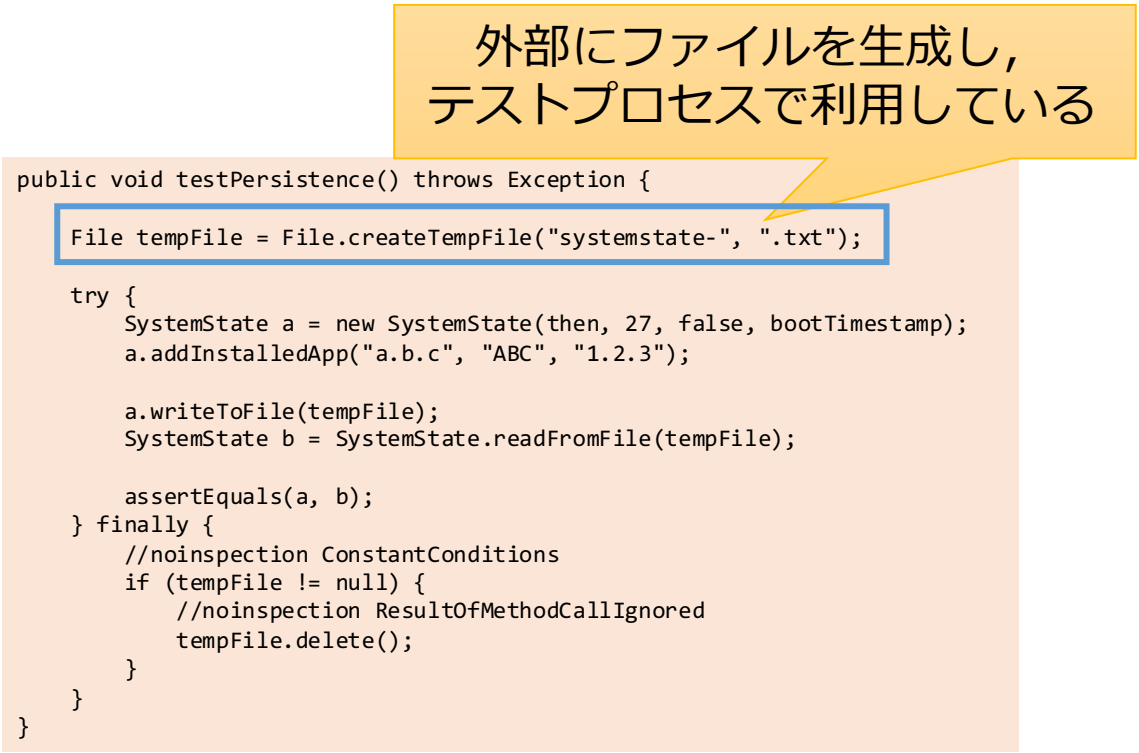

図 7 Mystery Guest の例

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見つけるのにあまり効果的でないと報告した．

Fabio Palomba[] らは，既存の自動生成ツールによって生成されるテストスイートは，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．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することを明らかにした．

自動生成ツールによって大量に生成されるテストスイートは，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．

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 [13] 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[] 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ぅ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．

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

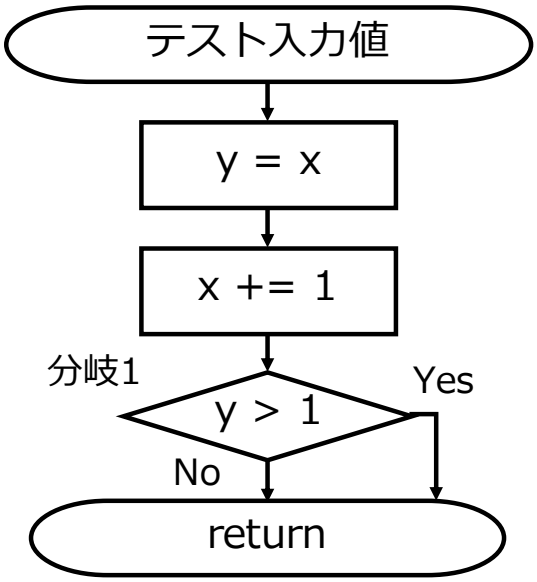

図 8 SBST によるテストケース生成の例

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 [14], [15], [16]．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提示させる．テストが失敗するたびに，開発者はテスト対象のプログラム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必要があるかどう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，自動生成によって得られる時間の節約よりも読みづらく，保守作業に助けになるというよりかむしろ邪魔するという結果が報告されている [1].

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. テストコード推薦手法

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，オープンソースソフトウェア (以下，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，推薦されるテストコード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．

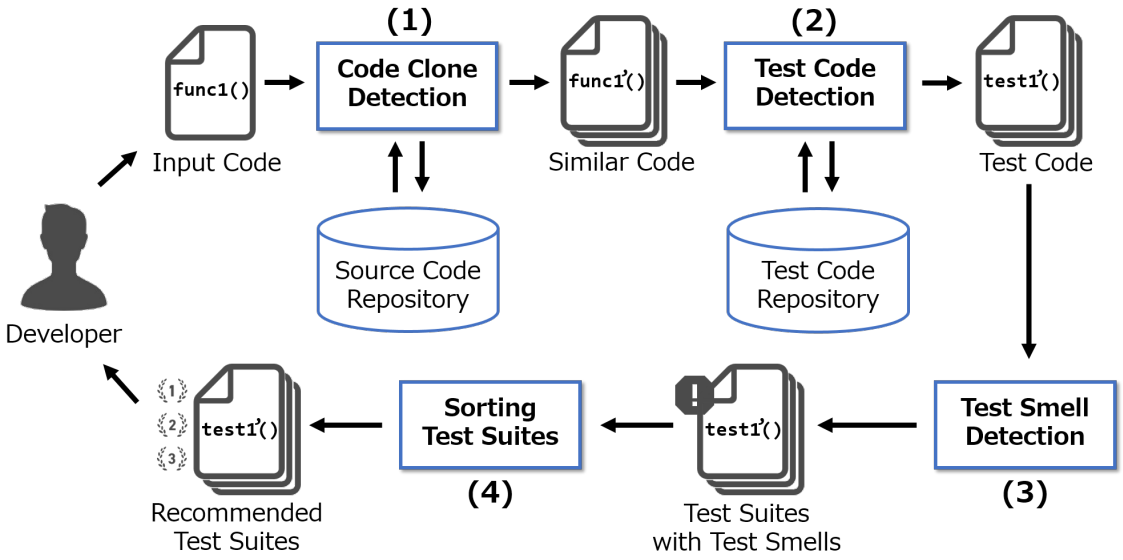

図 9　提案手法の概要

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，入力コードに対応する類似コードを検出する．

Step2

複数の類似コードが検出されると，次にその類似コード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と入力コードの類似度と 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キングされる．

以降の節で，それぞれのステップの詳細について説明する．

3.1 Step1: 類似コード検出

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，クローンペア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として扱う．

SuiteRec は，NiCad を用いて入力コード対応する類似コードを大規模な OSS プロジェクトを保持する Github 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，プロジェクト内に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．我々は，検索時間を短縮するために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複数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 の標準設定で提案ツールに実装した．

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 つのステップを実施する．

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
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．Step1 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．

Step2 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??の例のようにテストコード内でオブジェクトの生成が行い，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，テストコードリポジトリ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．しかし，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 Step3 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 [?]．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，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．

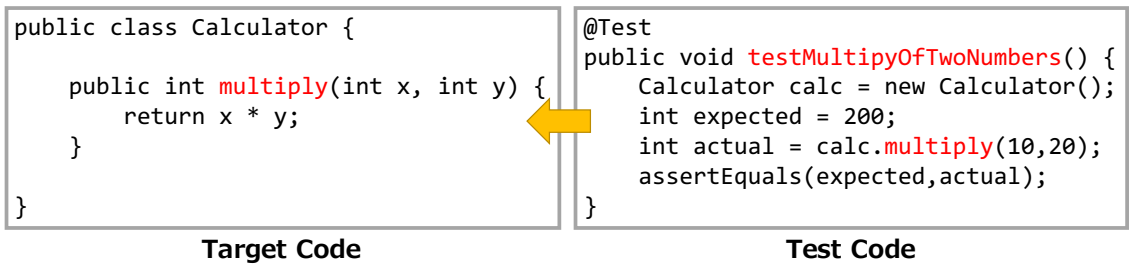

図 10　テストコードと対象コードの対応付け

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[?] 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正しく検出でき，多くの研究で利用されている [][][]．本研究では，tsDetect で検出できる 19 種類のテストスメルの内，2.4 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はな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: 推薦されるテストコードのランキング

最後の Step4 では，推薦されるテストスイートをランキングし，開発者が求める順番でテストスイートを提示する．

SuiteRec のランキングは，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 [fose2019] 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ンを対象に，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を関係を調査した．その結果，テストコードの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ている．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順番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 *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．

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

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．

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．

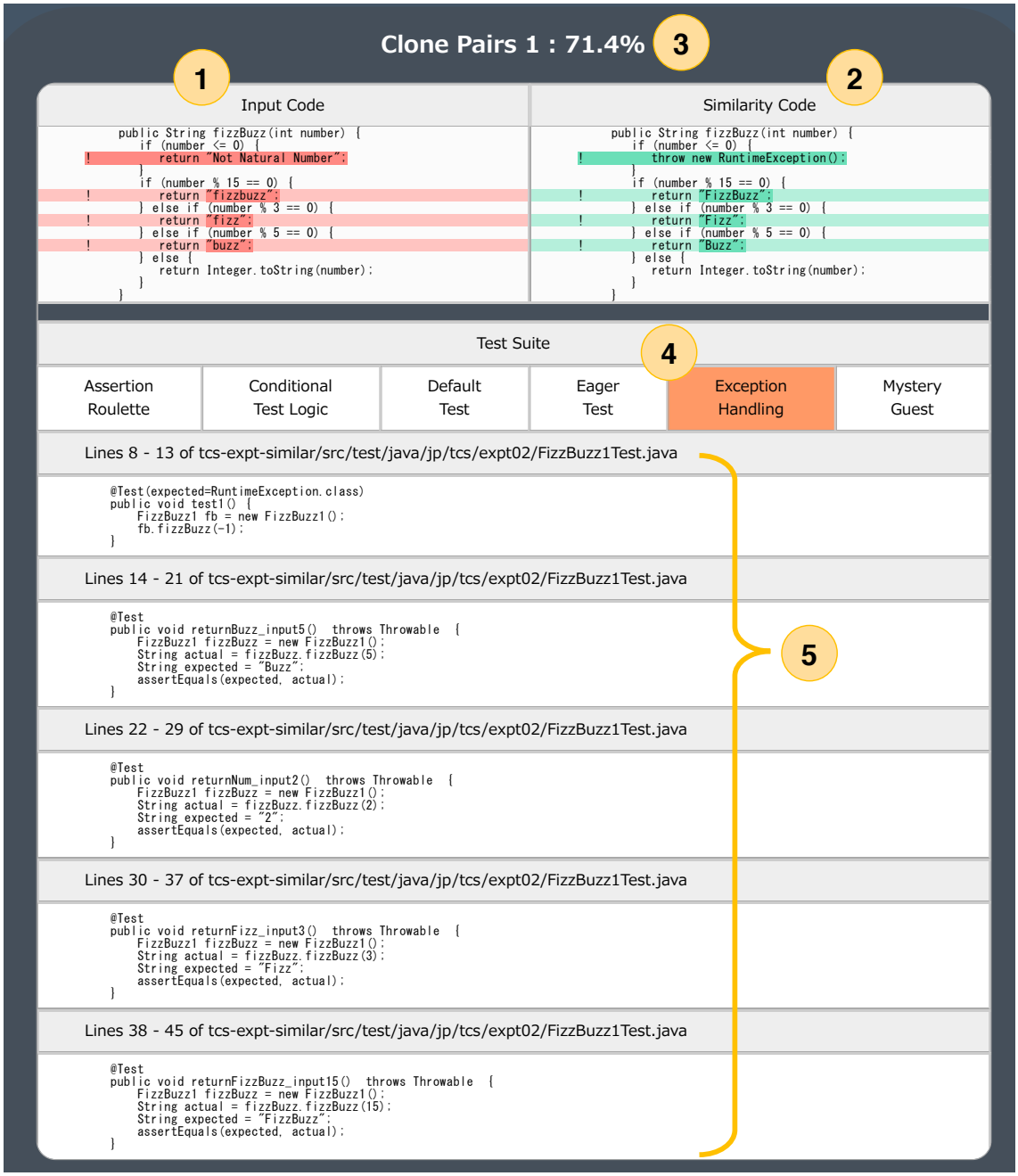

図 11 SuiteRec から推薦されるテストコード

(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，被験者による実験を行う．実験では，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のか？

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，高い品質を持つ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？

RQ5: SuiteRec は，開発者が求める順番でテストスイートをランキングできるか？

RQ5 では，SuiteRec から推薦されるテストスイートのランキングの妥当性を検証する．

以降，4.1 節では，本実験に参加した被験者について説明する．4.2 節では，被験者に実施してもらう実験タスクについて説明する．最後に 4.3 節で，実験手順について説明する．

4.1 被験者の選択

評価実験のために，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 人対して実験を実施した．事前アンケートによると 90%以上の学生が 2 年以上のプログラミング経験があり，80%以上の被験者が 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，80%以上が単体テストの作成経験があった．

4.2 実験タスクの作成

評価実験を実施するために，3 つの実験タスクを被験者に割り当てた．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，被験者が一番初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.

```java
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.

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.

```java
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

```
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

4.3　実験手順

4.1 節で説明した実験タスクを用いた被験者による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ツールの利用とテストコードの作成を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．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．

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

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	○

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．実験中，被験者は過去の回答を参考できないようにした．

4.4 実験結果

本節では，10 人の被験者による 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，5 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．

4.4.1 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スイートの命令網羅と分岐網羅の 2 種類のコード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[?] のプラグインとして搭載されている EclEmma[?] 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**??**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．

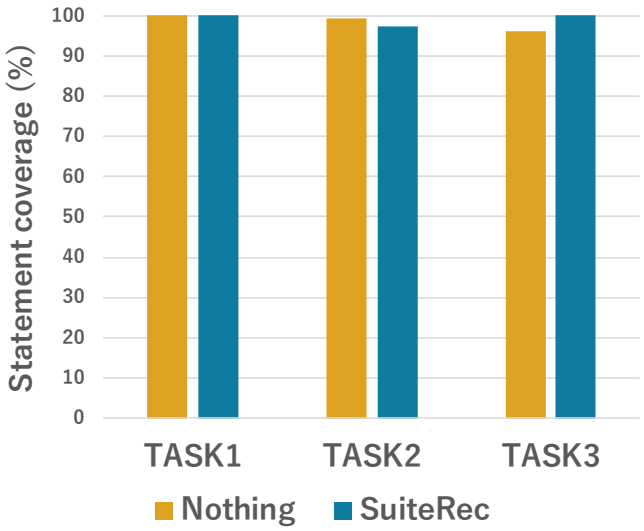

図 15　命令網羅の平均カバレッジ

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

4.4.2　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大きくなっていることが分かる．この結果は SuiteRec によって

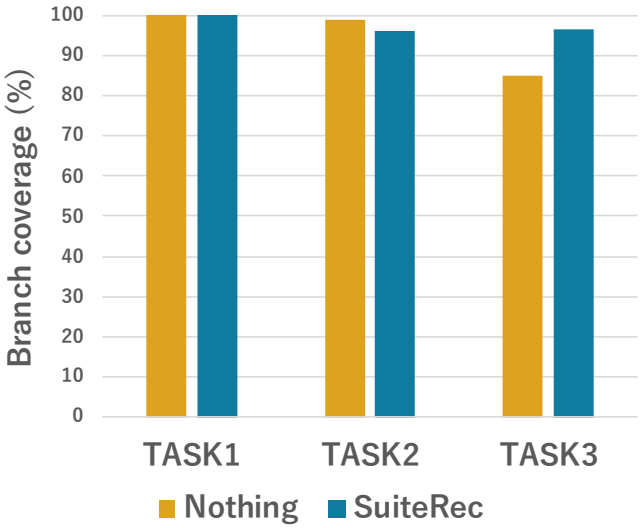

図 16　分岐網羅の平均カバレッジ

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 (テスト項目) 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．

SuiteRec の利用は，推薦されたコード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

4.4.3　RQ3: SuiteRec は，高い品質を持つ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．図の結果は，すべての被験

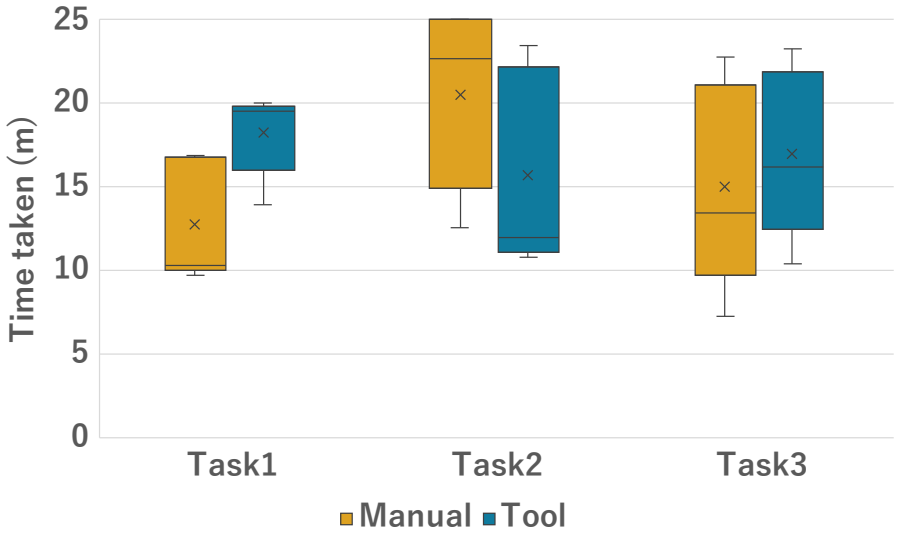

図 17　テストコード作成タスク終了までの時間

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．実際のアンケートの記述でも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．一方で，テストスメルが含まれていることは気づいていたがリファクタリングの方法が分からずそのまま提出したと述べている被験者も存在した．これは今後のツールの課題であり，テストスメルのリファクタリング方法も提示する改良の必要がある．

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 倍以上の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 [?]．

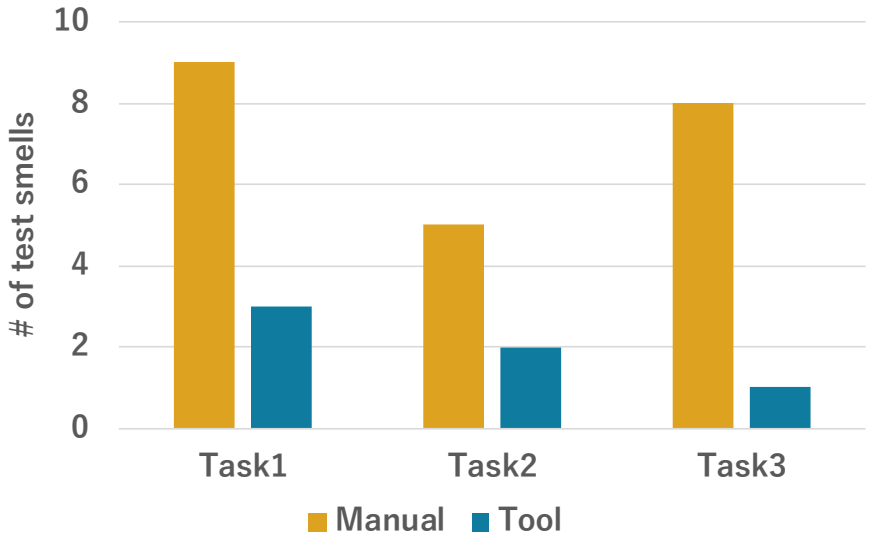

図 18　テストコード内に含まれていたテストスメルの数

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

4.4.4 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，この結果はこの結果は実際のタスクの終了時間と長さ (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終了時間が遅いことが分かる．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，それが後のメンテナンス活動を困難にする．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な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．

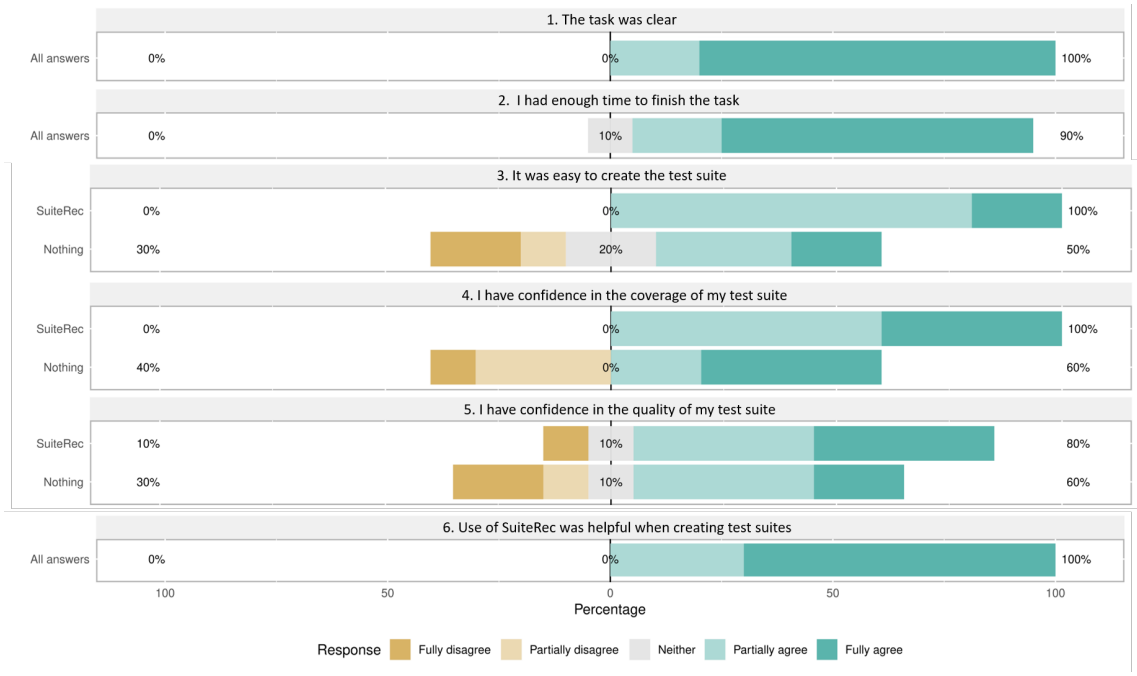

図 19　実験後のアンケートの回答

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

4.4.5　RQ5: SuiteRec は，開発者が求める順番でテストスイートをランキングできるか？

5. 議論

本章では，3 章のテストコード推薦手法及び 4 章の評価実験の結果について議論する．

5.1 SuiteRec の有用性

本節では，テストコード作成支援とテストコードの品質の観点から SuiteRec の有用性について考察する．

5.1.1 テストコード作成支援

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．実際にアンケートの自由記述欄の回答はこの考察を裏付けている．例えばある被験者は「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．

5.1.2 テストコードの品質

RQ3 は，SuiteRec を使用して作成したテストコードは品質が高いことを示した．開発者によって埋め込まれるテストスメルの存在は，メンテナンス活動を困難にする可能性がある．SuiteRec はテストスメルの存在を被験者に意識させることで，初めから理解しやすく良質なテストコードを作成できた．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な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．

6. 関連研究

本章では，提案手法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[1] ら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テストを再利用するツール Grafter を提案した．Grafter の評価実験では，コードクローン検出ツール DECKARD[17] によって有名 OSS プロジェクトから検出された 52 個のクローンペアの内 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，Gilligan[10]，[34] に基づいており、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．

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されるのは後のメンテナンス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