SuiteRec: Automatic Test Suite Recommendation System Using Code Clone Detection Tool

*Note: Sub-titles are not captured in Xplore and should not be used

1st Ryosuke Kurachi
Information Science
Nara Institute of Science and Technology
Nara, Japan
kurachi.ryosuke.kp0@is.naist.jp

2nd Eunjong Choi
dept. name of organization (of Aff.)
Kyoto Institute of Technology
Kyoto, Japan
echoi@kit.ac.jp

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—It is important to support software testing to ensure software quality. In previous studies, various automatic generation techniques have been proposed to reduce test creation costs. However, automatically generated tests are usually not based on the process and intention of creating the target code, and are therefore generally considered to be less readable. And it makes later maintenance activities difficult. This places a question mark over their practical value. In this research, we propose SuiteRec, a tool that recommends existing high quality test codes that exist on OSS to solve this problem. SuiteRec considers test reuse between clone pairs using similar code search technology. SuiteRec detects similar codes from the input code and recommends a test suite corresponding to the similar codes to the developer. Further, SuiteRec shows the developer a test smells that means a bad implementation of the test code. and the recommendation ranking is ranked so that a higher quality test suite can be recommended. In the evaluation of the proposed tool, the test code was created depending on whether the subject used SuiteRec or not, and the difference was compared. .With various experiments, we show that (1) it is effective to increase code coverage when creating test code for programs with many conditional branches, (2) The test code created using SuiteRec has a high quality with a small number of detected test smells, (3) When using SuiteRec, developers feel that it is easy to create test code, and they can be confident in the created code.**

***Index Terms*—clone detection, recommendation system, software testing, unit test**

I. Introduction

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，現状では単体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[2] 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 [3]．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．

本研究では，この課題を解決するために 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 を紹介します．SuiteRec は類似コード検出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ます．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します．さらに，テストコードの良くない実装を表すメトリクス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ています．

提案ツールの評価では，被験者によって SuiteRec の使用

Identify applicable funding agency here. If none, delete this.

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，提案ツールの利用は分岐が多く複雑なプログラムのテストスイートを作成する際に，コードカバレッジを向上させることができることや，ツールを使用して作成テストコードの品質が高いことが分かった．また，定性的な評価として実験後にアンケートを実施し，推薦ツールを使った場合多くの被験者は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．

II. BACKGROUND AND RELETED WORK

Unit testing. 単体テストの実行タスクでは，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を確認する．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 などのテスト自動実行ツールの利用が産業界で進んでいる．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．

単体テスト設計タスクで作成されるテストケースは，テスト手順，テスト入力値，テスト期待結果から構成される．テスト手順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結果と比較する．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．

Test case generation. 既存の研究 [4] は，既存のテストケースを再利用，自動生成，または再適用できることによって，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．

RT とは，ソフトウェアにランダムな入力を与えるテスト手法である．無造作・均一にテストを実行するランダムテストは自動化に適しているが，コードカバレッジ率向上，バグ検出の観点において，テストケース 1 件当たりの効率は著しく悪い．

SE は対象コードを静的解析してプログラムを記号値で表現し，コード上のそれぞれのパスに対応する条件を抽出し，パスごとにパスを通るような入力値が満たすべき条件を集める．そして，パスごとにその条件を SMT ソルバ [5] などの制約ソルバを用いて解き，得られた具体値をテスト入力値とする．

SBST は，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ように設計した評価関数に基づいて，ヒューリスティック探索アルゴリズムを用いて達成したい要件を満足するテストスイートを生成する技術の総称である．

MBT はモデルに基づいてテストスイートを生成する技術の総称である．モデルは何らかの形でテスト対象を記述したものであり，要求分析や設計のためのモデルを活用することもあれば，テストのためにモデルを作成することもある．

CT は，パラメータ間の相互作用に起因する不具合を効果的に発見するためにテストケースとしてパラメータに割り当てる値の組み合わせを生成する手法である．

Test Smell. プロダクションコードだけでなく，テストコードのも適切なプログラミングの慣習に従って設計する必要があります [44]．テストコードのを適切に設計することの重要性は元々Beck[7] によって提唱されました．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技術を定義しました．このカタログはそれ以降，18 個の新しいテスト臭を定義した Meszaros [42] によってより拡張されました。最近の研究では，テストスメルの存在は開発者のテストスイートの理解に悪い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くなると言われています．

III. SUITEREC

SuiteRec takes a code fragment of a function unit from a developer as input code and searches for similar codes of the input code. Then, test suites corresponding to similar codes are sorted and presented to developers in order of priority.

Figure 1 shows the flow until a test suite is recommended by SuiteRec. The recommendation method mainly consists of the following 4 steps.

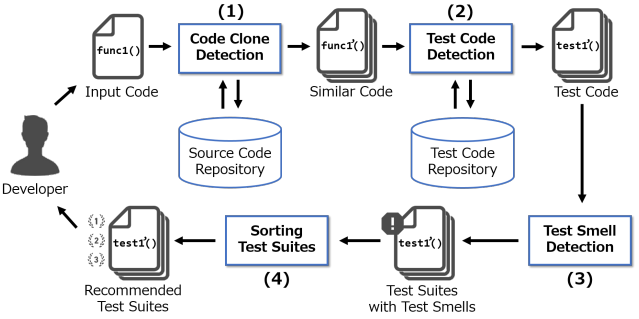

Fig. 1. Overview of SuiteRec.

(1) When SuiteRec receives a input code, it searches the source code repository for the corresponding similar code fragments using a existing code clone detection tool.
(2) Detected similar code fragments, SuiteRec searches the test code repository for the test suite corresponding to the similar code.
(3) SuiteRec detects test smells in the test suite collected by the previous step using existing test sumel detection tools.
(4) As the final step, SuiteRec sorts test suites in descending order of priority based on similarity and number of test smells.

A. Code Clone Detection

In this study, NICAD [3] was adopted as a similar code detection tool. NICAD converts code fragment layouts uniformly and detects code pairs by comparing code fragments

in units of functions. By adopting such a method, NICAD has realized clone pair detection with high accuracy and high recall. NICAD searches the Github repository hosting large open source projects for similar code corresponding to the input code.

The source code repository in Fig. 1 contains only the production code of the Github project with test code. Specifically, we selected a project that had a test folder in the project and adopted the JUnit testing framework. NICAD has a project size limit that can be searched at once. In order to shorten the search time, large-scale projects were divided, small-scale projects were integrated, and multiple search processes were run in parallel, making it possible to search for similar codes in real time. The detection setting is implemented in the proposed tool as a standard setting of NICAD.

B. Test Code Detection

In order to search for test suites corresponding to similar code fragments, the target code is associated with the test code. In this research, the following two steps are taken in order to precisely associate the test code with the target code.

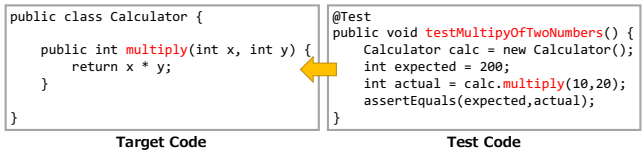

Fig. 2. Example of mapping test code to target code.

(1) Static analysis of test code and confirmation of method calls.
(2) Divide the test method with a delimiter or capital letter and associate it when the target method partially matches.

In the unit test, an object is generated in the test code as shown in the figure 2, and it is executed by calling a method of the test target code. Therefore, the test code in the test code repository is statically analyzed and the method call is obtained to associate the test target code with the test code. However, multiple methods may be called in the test method, so the method names are also compared. It is recommended to faithfully represent the contents of the processing of the target method as the test method name description method, and the name of the target method is often described in the test method name. Therefore, the name of the test method is divided by a delimiter or capital letter, and it is linked if it partially matches the target method.

The test code repository in Figure 1 stores test code corresponding to the production code in the source code repository. As a pre-processing, static analysis was performed on a large-scale project in advance, and information that linked production code and test code was stored in the DB, so that test code could be searched at high speed via the DB.

C. Test Smells Detection

In this study, tsDetect [6] was adopted as a test sumle detection tool. tsDetect is a tool implemented with an AST-based detection method that can detect 19 test smells. It has also been reported that test smells can be detected correctly with 85% to 100% accuracy and 90% to 100% recall. In this study, we implemented the following 6 types of test smells, which are important in considering the recommendation of test codes among 19 test smells that can be detected by tsDetect.

TABLE I
SUBJECT TEST SMELLS

Name	Description
Assetion Roulette	Occurs when a test method has multiple non-documented assertions. Multiple assertion statements in a test method without a descriptive message impacts readability/understandability/maintainability as it's not possible to understand the reason for the failure of the test.
Conditional Test Logic	Test methods need to be simple and execute all statements in the production method. Conditions within the test method will alter the behavior of the test and its expected output, and would lead to situations where the test fails to detect defects in the production method since test statements were not executed as a condition was not met. Furthermore, conditional code within a test method negatively impacts the ease of comprehension by developers.
Default Test	Test code in which the test class or test method name is the default in test code using a testing framework such as JUnit. It is necessary to change the name appropriately to improve the readability of the code.
Eager Test	Occurs when a test method invokes several methods of the production object. This smell results in difficulties in test comprehension and maintenance.
Exception Handling	This smell occurs when a test method explicitly a passing or failing of a test method is dependent on the production method throwing an exception. Developers should utilize JUnit's exception handling to automatically pass/fail the test instead of writing custom exception handling code or throwing an exception.
Mystery Guest	Occurs when a test method utilizes external resources (e.g. files, database, etc.). Use of external resources in test methods will result in stability and performance issues. Developers should use mock objects in place of external resources.

In addition, the test code including the following four test smells that are not suitable as recommended test code has been deleted from the test code repository in advance, so that it is not output as a recommended test code.

- **Empty Test.**Occurs when a test method does not contain executable statements.
- **Ignored Test.**Test code that has the @Ignore annotation and is not executed.
- **Redundant Assertion.**This smell occurs when test methods contain assertion statements that are either always true or always false.

- **Unknow Test.**A test method that does not contain a single assertion statement and @Test(expected) annotation parameter.

D. Sort Recommended Test Suites

The recommended test suites were ranked based on the similarity between the input code and the detected similar code and the number of test smells included in test suites. We investigated the relationship between the similarity between clone pairs and the similarity between test code pairs for clone pairs with test code in both code fragments on OSS.

As a result, there was a correlation between the similarity between the test code pairs and the similarity of the target clone pair. Therefore, we consider that the clone pairs with higher similarity between the input code and the similar code are easier to reuse the test code.

SuiteRec implements a recommendation ranking that sorts the clones in the order of high similarity and determines the order based on the number of test sumels when the similarities are the same.

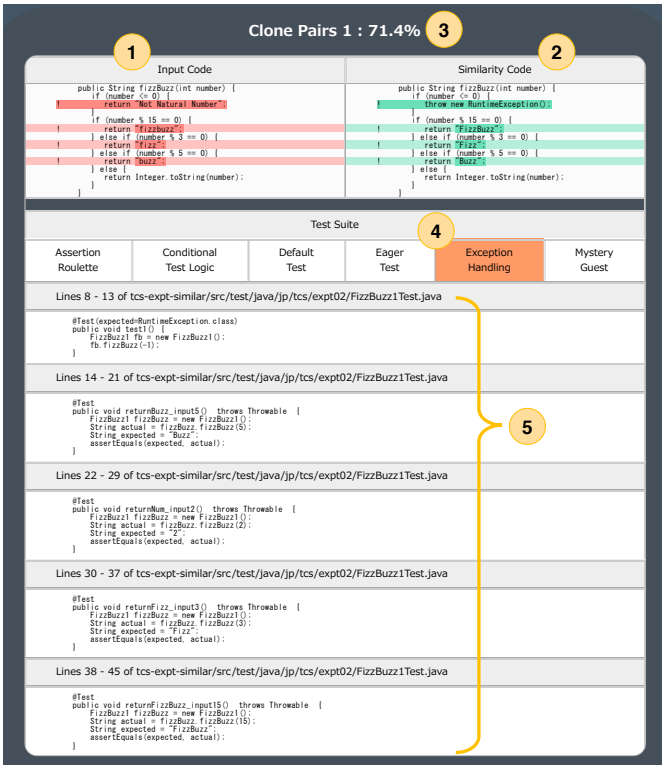

Fig. 3. Test suite recommended by SuiteRec.

(1) **Input Code.**The target code entered by the developer is displayed.
(2) **Similarity Code.**A similar code for the input code is displayed. The differences are highlighted so that you can see the difference between the input code and the similarity code.
(3) **Degree of similarity.**The similarity between the input code and the similar code is displayed. The similarity is calculated using the Unique Percentage of Items (UPI) method used by NICAD[?].
(4) **Test Smells.**If test smells are included in the test suite, the test smell is highlighted in orange, and the developer is presented with the presence of test smells.
(5) **Recommend Test Suites.**The recommended test suite is displayed. A file path is also displayed to indicate from which project the test code was referenced.

IV. Evaluation

In this section, we will conduct experiments with subjects to evaluate SuiteRec quantitatively and qualitatively. Subjects will be asked to create test codes for three production codes. Evaluate SuiteRec by comparing the test code with and without using SuiteRec.

By collecting data on code coverage, time to complete experimental tasks and test code quality throughout the experiment, we aim to answer the following research questions:

- RQ1: **Can SuiteRec support the creation of tests with high coverage?** Coverage is an important factor as an indicator of software quality. If there is a line that is never executed in the test code, the quality of that part cannot be ensured. Can SuiteRec help increase coverage?
- RQ2: **Can SuiteRec reduce test code creation time?** Can developers shorten test code creation time by referring to test codes recommended by SuiteRec?
- RQ3: **Can SuiteRec support high quality test creation?** Can developers create high-quality test code by referring to the test code recommended by SuiteRec?
- RQ4: **How do using SuiteRec influence the developers' perception of test code creation tasks tasks?** Do developers find it easier to create test code when using SuiteRec, and are they more confident in their created test code?

A. Participant Selection

We recruited students with basic programming skills and an understanding of software testing. The experiment was conducted with 10 master students who majored in information science.According to the preliminary questionnaire, more than 90% of the students had more than 2 years of programming experience, and more than 80% of the subjects had more than 1 year of Java language experience. All students had basic knowledge about software testing in lectures and other lectures, and more than 80% had experience creating unit tests.

B. Object Selection

To conduct the experiment we prepared three production codes. It is assumed that the subjects fully understand the specifications of production code in order to create test code. Therefore, we selected a typical computational problem that often uses competitive programming as production code. In addition, a specification written in natural language was prepared so that the specification of the production code could be confirmed. In order to make a difference in each task, the

number of conditional branches in each task was increased to 8, 16, and 24.

Figure 4 is an example of the production code that was presented. In the post-experimental questionnaire, it was confirmed that all the subjects expressed a positive opinion about the understanding of the experimental task. Also, there was no negative answer to the question about whether there was enough experiment time. Therefore, it can be seen that the subject fully understood the given experimental task and had sufficient work time.

```
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";
        }
    }
}
```

Fig. 4. Example of a experimental task.

C. Experiment Procedure

First, we conducted a 30-minute lecture and practice task on using JUnit from basic knowledge about software testing, and confirmed understanding of the test code description. And we asked them to create test codes for the three production codes for the actual experiment.

Ask the subjects to judge the end of the experimental task. Specifically, the test task was completed when the subjects were satisfied with the coverage and quality of the test code they created. The experiment time was a maximum of 25 minutes per task.

To prevent the use effect of SuiteRec from being biased by tasks, subjects were assigned to change whether or not SuiteRec was used depending on the task. In order to prevent the learning effect when SuiteRec is used, tasks are assigned so that SuiteRec is not used continuously in three tasks. The subjects were not allowed to refer to past answers.

V. Results

In this section we present the quantitative and qualitative evaluation results of SuiteRec by 10 subjects, as described in the previous section, for each of the research questions.

A. RQ1: Can SuiteRec support the creation of tests with high coverage?

In this experiment, we calculated two types of code coverage: statement coverage(C0) and branch coverage(C1) of test suites submitted by the subjects. To calculate the coverage, we used EclEmma[?], which is installed as a plug-in of the integrated development environment Eclipse[?]. Figures 1 and 2 show the average coverage of statement coverage and branch coverage, respectively. As a result, there is almost no difference in the coverage rate of statement coverage in all three tasks depending on whether SuiteRec is used or not, and the coverage of each task exceeds 90%.

Regarding the branch coverage in Fig. 2, it can be seen that there is almost no difference between TASK1 and TASK2 with 8,16 branches depending on whether SuiteRec is used or not.

However, the results of TASK3 with the largest number of branches showed that there was a difference of more than 10% in the average coverage of the subjects.

This result suggests that the test code recommended by SuiteRec is useful for increasing the coverage rate when creating test code for production code with many branches. In fact, in the questionnaire after the experiment, there were multiple reports that the subjects were able to follow the test items that were overlooked by the recommendation code.

Fig. 5. Statement coverage (C0).

Fig. 6. Branch coverage (C1).

B. RQ2: Can SuiteRec reduce test code creation time?

Figure 5 shows the results of comparing the time spent completing the test code creation task with and without SuiteRec.

It can be seen that the test creation time is longer when SuiteRec is used for two tasks than when it is not.

This result can take time to read and understand multiple test suites recommended by SuiteRec.

Subjects cannot reuse the recommended test code as is.

It is necessary to rewrite the test code by looking at the difference between the input production code and the detected similar code.

図 5 は，SuiteRec を使用した場合と何も使用しない場合で，テストコード作成タスクの終了までに費やされた時間を比較しています．3 つの問題の内，2 つの問題で SuiteRec を使用した場合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ります．被験者は，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ません．入力したプロダクションコードと検出された類似コードの差分を見てテストコードを書き換える必要があります．また，実験後のアンケートではテスト対象のオブジェクト生成の記述を再利用する際にその都度書き換える必要があり，時間がかかってしまったと述べている．問題 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 (項目) の数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無駄な時間を費やしてしまった可能性がある．

Fig. 7. Time taken to create test code.

C. RQ3: Can SuiteRec support high quality test creation?

図 6 は SuiteRec を使用した場合とそうでない場合で，提出されたテストコード内のテストスメル数を比較しています．すべての TASK 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，推薦されるテストコード自体の品質が高く開発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したと考えられる．また，ツール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より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 [6].

Fig. 8. Number of detected test smells.

D. RQ4: How do using SuiteRec influence the developers' perception of test code creation tasks tasks?

図 7 は，実験後のアンケートの回答の結果をまとめたものです．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ことが分かります．

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ます．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 2) 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ります．被験者は，推薦された複数提案される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します．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ります．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ます．アンケートによるツールの改善点への自由記述では，テストコードの編集作業を支援する機能 (クラス名やメソッド名を入力コード対応する名前に自動編集する機能など) を追加した方が良いという多くの意見を頂来ました．SuiteRec の更なる改善は，実験タスクの完了時間を短縮できる可能性を示しています．

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 (質問 5)．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ている．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っています (図 3)．自身が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す．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す．

被験者は，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ません．実際の提出されたテストコード内のテストスメルの数も SuiteRec を使わなかった場合は，使った場合と比べて多く存在していることが分かります (図 4)．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させます．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します．一方で，SuiteRec を利用した場合でも品質に関してネガティブ

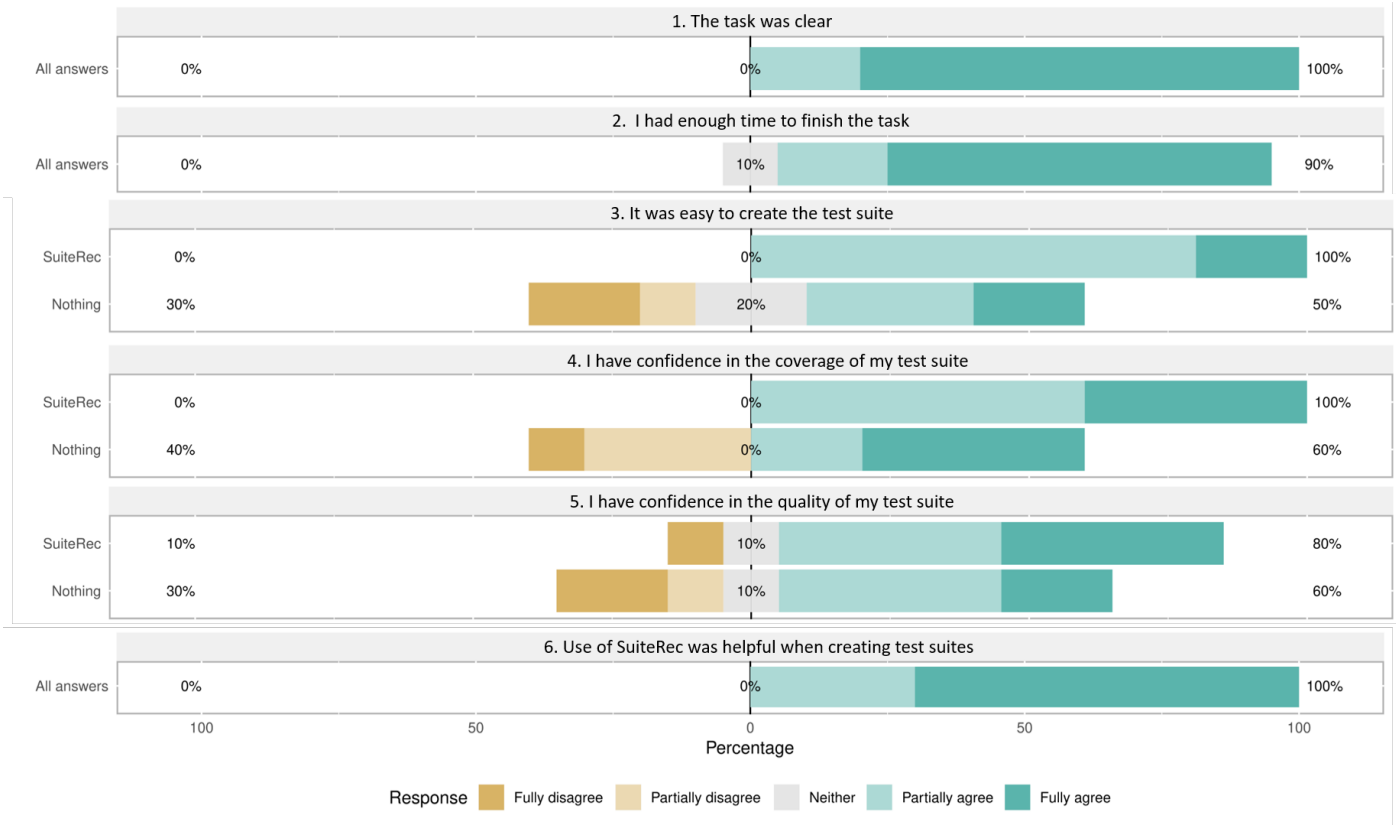

Fig. 9. キャプション

な意見も存在します．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ます．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．

VI. Related work

Code recommendation. コード推薦システムは，他のプログラムのコードフラグメントを提示し再利用できるようにしたりすることで開発者を支援します．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手法を提案している．Mostafa[2] らは，自身のプロジェクトだけでなく他のプロジェクトを横断してクローンペアを検出しテストコード再利用することの有効性を調査した．

VII. Conclusion and Future Work

SuiteRec は，ユーザーが入力した関数単位のプロダクションコードに対して，類似コード検出ツールを用いて OSS 上に存在する既存のテストコードを推薦するツールです．さらに，テストコードの良くない実装を表すメトリクス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ています．分岐が多くテスト項目の作成が難しいプロダクションコードに対して，SuiteRec を使用してテストコード作成するとカバレッジを向上できる可能性があります．また，品質の高いテストコードを作成でき，開発者は自分で書いたコードに自信が持つことができます．今後の課題としては，より実践的な利用に備えてツールを改善する必要があります．さらに SuiteRec が推薦するテストスイートの優先順位に対する妥当性評価も実施する予定である．

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
[4] K. Elissa, “Title of paper if known,” unpublished.
[5] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
[7] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.