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

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

2nd Eunjong Choi
Kyoto Institute of Technology
Kyoto, Japan
echoi@kit.ac.jp

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

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

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

6th Given Name Surname
dept. 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 sorted 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) SuiteRec is effective to increase code coverage when creating test code for programs with many conditional branches, (2) Using SuiteRec takes a lot of time to create test code, (3) The test code created using SuiteRec has a high quality with a small number of detected test smells, (4) 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．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 [20]．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 [3]，[16]，[17]，[18]，[19].

既存研究で提案されている EvoSuite [3] 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 [13]，[14]，[15]．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 [1].

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.

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

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 [12] 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 [4]．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 [5] によって提唱されました．さらに，Van Deursen ら [7] 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 [6] によって拡張されました。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 [8]．

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.

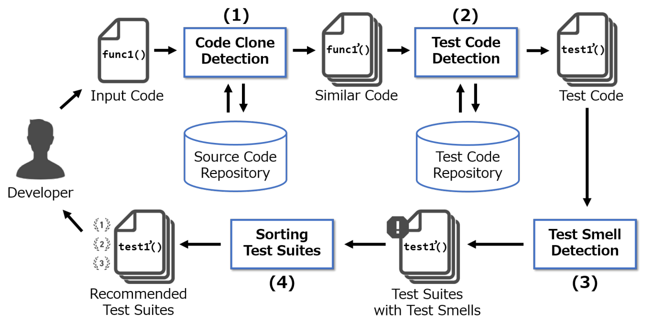


Fig. 1. Overview of SuiteRec.

(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.

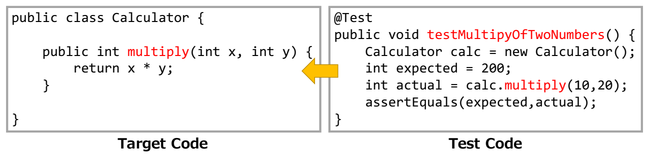
A. Code Clone Detection

In this study, NICAD [2] 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.



```
public class Calculator {

    public int multiply(int x, int y) {
        return x * y;
    }

}
```

```
@Test
public void testMultipyOfTwoNumbers() {
    Calculator calc = new Calculator();
    int expected = 200;
    int actual = calc.multiply(10,20);
    assertEquals(expected,actual);
}
```

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 [10] 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 [9].

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.

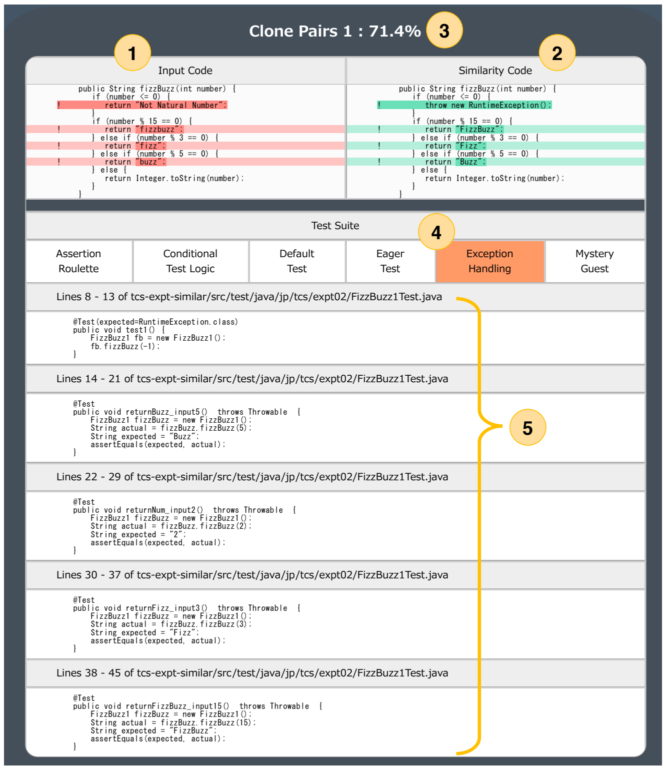
- **Empty Test.** Occurs when a test method does not contain executable statements.
- **Ignored Test.** Test code that has the @Ignore annotation and is not executed.
- **Redundant Assertion.** This smell occurs when test methods contain assertion statements that are either always true or always false.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



```
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);
    }
}
```

```
public String fizzBuzz(int number) {
    if (number <= 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);
    }
}
```

Lines 8 - 13 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java

```
@Test(expected=RuntimeException.class)
public void test1() {
    FizzBuzz1 fb = new FizzBuzz1();
    fb.fizzBuzz(-1);
}
```

Lines 14 - 21 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java

```
@Test
public void returnBuzz_input5()  throws Throwable  {
    FizzBuzz1 fizzBuzz = new FizzBuzz1();
    String actual = fizzBuzz.fizzBuzz(5);
    String expected = "Buzz";
    assertEquals(expected, actual);
}
```

Lines 22 - 29 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java

```
@Test
public void returnNum_input2()  throws Throwable  {
    FizzBuzz1 fizzBuzz = new FizzBuzz1();
    String actual = fizzBuzz.fizzBuzz(2);
    String expected = "2";
    assertEquals(expected, actual);
}
```

Lines 30 - 37 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java

```
@Test
public void returnFizz_input3()  throws Throwable  {
    FizzBuzz1 fizzBuzz = new FizzBuzz1();
    String actual = fizzBuzz.fizzBuzz(3);
    String expected = "Fizz";
    assertEquals(expected, actual);
}
```

Lines 38 - 45 of tcs-expt-similar/src/test/java/jp/tcs/expt02/FizzBuzz1Test.java

```
@Test
public void returnFizzBuzz_input15()  throws Throwable  {
    FizzBuzz1 fizzBuzz = new FizzBuzz1();
    String actual = fizzBuzz.fizzBuzz(15);
    String expected = "FizzBuzz";
    assertEquals(expected, actual);
}
```

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?

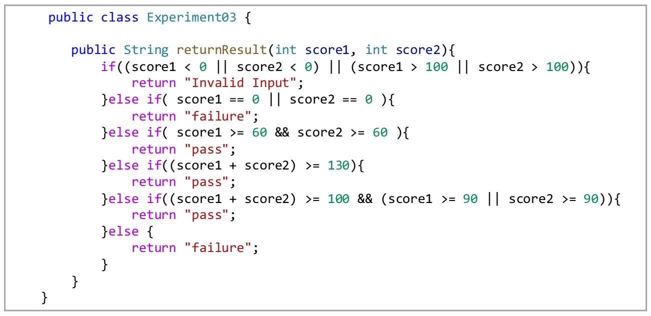
A. *Participant Selection*

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.

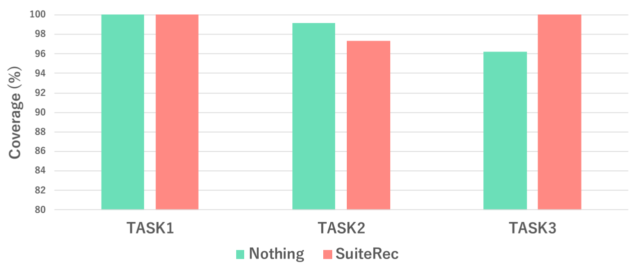


Fig. 5. Statement coverage (C0).

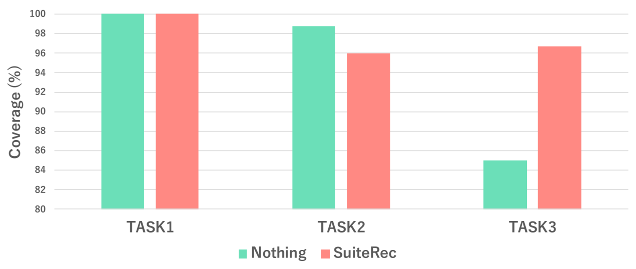


Fig. 6. Branch coverage (C1).

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 tasks 1 and 3 than when it is not. This result can take time to read and understand multiple test suites recommended by SuiteRec. Subjects will not be able to reuse the recommended test code without modification. It is necessary to rewrite the test code by looking at the difference between the input production code and the detected similar code. In addition, according to the questionnaire after the experiment, it was necessary to rewrite each time the object creation sentence was reused, and it took time.

For Task 2, it can be seen that the test creation time is shorter when SuiteRec is used. We examined the submitted test code and found that there were many test cases (items) when SuiteRec was not used, although there was no difference in coverage. This result suggests that the subjects may have wasted time creating many useless test cases.

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

Figure 6 shows the results of comparing the number of test smells in the submitted test code with and without SuiteRec. For all TASKs, the test code created using SuiteRec contains less test smell than if it were not used. This result suggests that the quality of the recommended test code is high, and the developer can create the test code while maintaining the

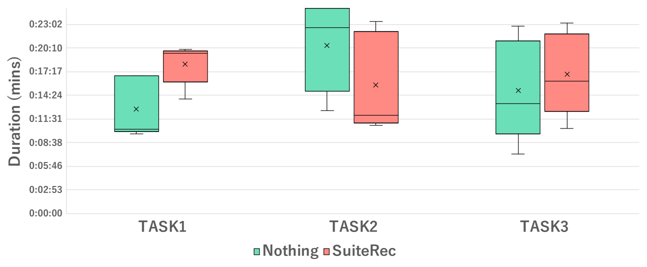


Fig. 7. Time taken to create test code.

quality by reusing it. Also, by presenting the test smells included in the recommended test suite, the test code may be rewritten based on it and a high quality test code may have been submitted. In the actual questionnaire responses, it was reported that the test smells presented were understood and refactored to eliminate them.

On the other hand, some subjects were aware that test smells were included, but did not know how to refactor. This is a topic for the future and needs to be improved to show how to refactor test smells.

When SuiteRec was not used, the subjects embedded more than five times the test smells compared to the case where it was used. This is probably because many subjects did not rename the default test method and wrote the Assert statement by copy and paste within one test method. In fact, it has been reported that many of these test smells are detected in existing projects [6].

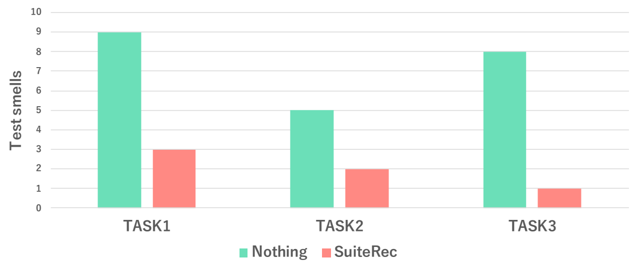


Fig. 8. Number of detected test smells.

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

Figure 7 summarizes answers to the survey questions. Overall, the subjects found the task clear (Question 1), and the allocated time sufficient (Question 2). For the remaining questions, you can see that there is a difference in opinion on the experimental task with and without SuiteRec.

When creating test codes, subjects can easily feel test code creation using SuiteRec. However, this result contrasts with the actual task end time and length (Figure 2), and it can be seen that the task end time is slower when SuiteRec is used. The subjects read and understand the recommended multiple suggested test suites and decide whether to reuse them. In addition, the test code cannot be applied without editing, and it is necessary to understand the difference between the input code and the detected similar code and make appropriate modifications to the test code. We speculate that when using SuiteRec, subjects may spend a lot of time on this part.

Many opinions that it is better to add a function that supports test code editing work (such as a function that automatically edits the class name and method name to the name corresponding to the input code) in the free description of the tool improvement by questionnaire I received. Further improvements in SuiteRec show the potential for reducing the completion time of experimental tasks.

When using SuiteRec, the subject is confident in the coverage of the test code created by himself (Question 5). On the other hand, 40% of subjects reported a negative response when nothing was used. However, it is known that there is almost no difference in the coverage of the test code actually submitted (Figure 3). It is important to be confident in the coverage of the test code you create. One of the purposes of software testing is that developers are responsible for the code they write and can provide software to users without anxiety.

When the test code was created without using SuiteRec, 40% of the subjects were not confident in the quality of the test code they wrote. It can be seen that the number of test smells in the actual submitted test code is greater when SuiteRec is not used than when it is used (Figure 4). Developers unknowingly embed test smells that make later maintenance activities difficult. The use of SuiteRec reduces the number of test mells by giving developers an awareness of the quality of the test code and brings confidence to the code they have created.

On the other hand, even when SuiteRec is used, there are negative opinions about the quality of the test code. The respondents reported that they understood Test Smells but didn't know how to refactor. This indicates the need for further improvement of SuiteRec, and we should add the function to present refactoring methods for each test smell.

VI. Conclusion and Future Work

SuiteRec is a tool that recommends existing test codes that exist on OSS using a similar code detection tool for the input production code. In addition, SuiteRec presents the test smell, an indicator of poor test code implementation, to the developer, and the order of recommendation is sorted so that higher quality test suites can be recommended.

According to the evaluation of SuiteRec by 10 subjects, it may be possible to improve coverage by creating test code using SuiteRec for production code which there are many branches and it is difficult to create test items. However, it takes a lot of creation time. Using SuiteRec also allows you to create high quality test code that doesn't contain much test smells, and developers can be confident in the code they write.

As a future task, SuiteRec needs to be improved for more practical use. Specifically, we are considering an automatic test code rewriting function. In addition, the validity of test suite priorities recommended by SuiteRec will be evaluated.

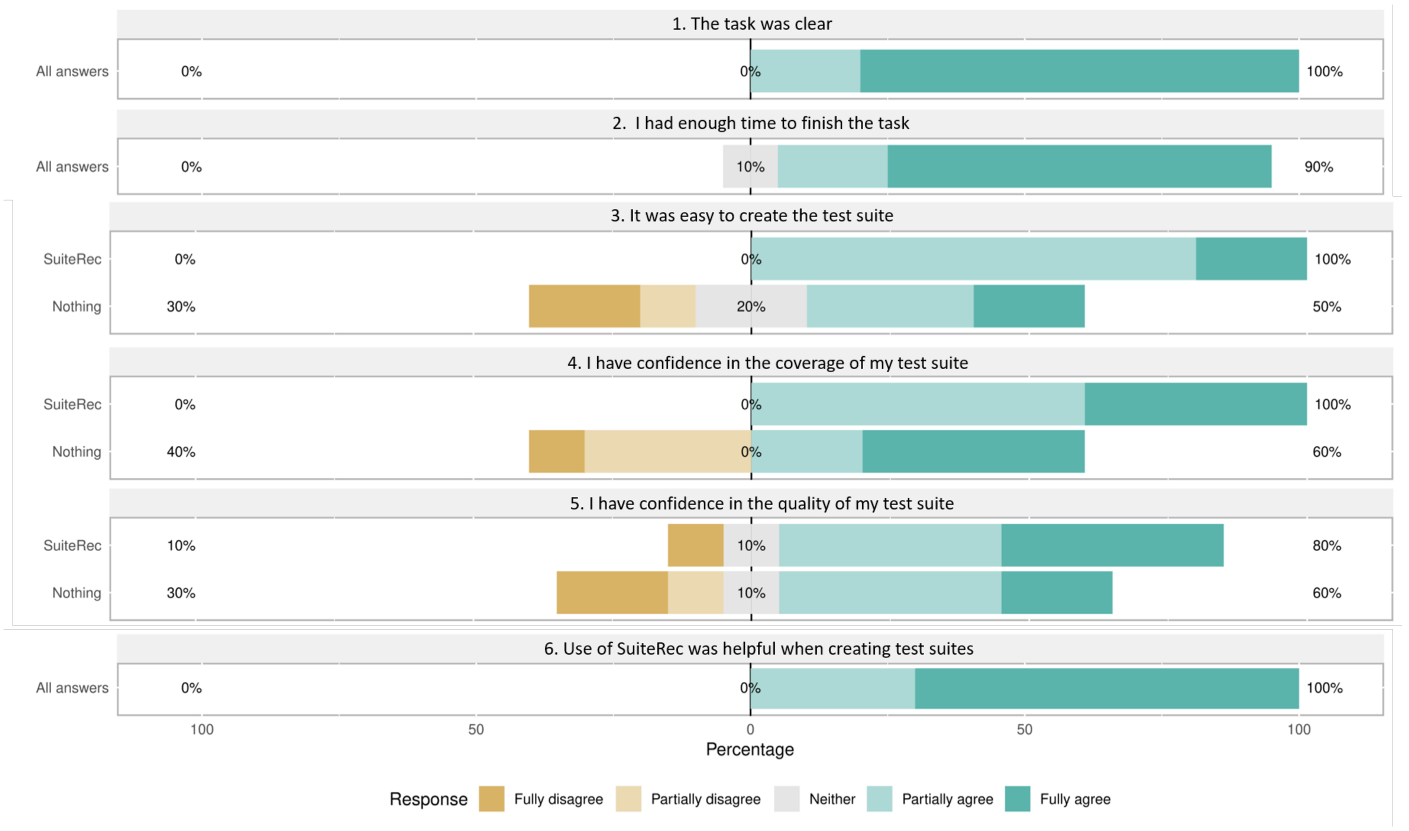


Fig. 9. キャプション

References

[1] S. Shamshiri, J. M. Rojas, and J. P. Galeotti, "How Do Automatically Generated Unit Tests Influence Software Maintenance?," In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pp.239–249, 2018.

[2] 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)*, pp.172–181, 2008.

[3] 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)*, pp. 416–419, 2011.

[4] N. Koochakzadeh and V. Garousi, "Tecrevis: a tool for test coverage and test redundancyvisualization," Testing–Practice and Research Techniques, 2010.

[5] M.Greiler, A.Zaidman, A.v.Deursen, and M.-A.Storey, "Strategies for avoiding text fixture smells during software evolution," In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pp. 387–396, 2013.

[6] G. Meszaros. xUnit Test Patterns: Refactoring Test Code. Addison Wesley, 2007.

[7] A. van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, pp. 92–95, 2001.

[8] D. Spadini, F. Palomba, A. Zaiaman, 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)*, pp. 1–12, 2018.

[9] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, and A. Ouni, 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)*, pp. 193–202, 2019.

[10] S. U. T. Smells. http://testsmells.github.io/, 2018.

[11] https://www.eclemma.org/

[12] P. Machado and A. Sampaio, "Automatic Test-Case Generation," Testing Techniques in Software Eng., vol. 6153, pp. 59–103, 2010.

[13] 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)*, pp. 547–558, 2016.

[14] 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)*, pp. 107–118, 2015.

[15] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: what if test code quality matters?," In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 130–141, 2016.

[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)*, pp. 212–223, 2015.

[17] A. Panichella, F. 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)*, pp. 1–10, 2015.

[18] I. W. B. Prasetya. "T3, a Combinator-Based Random Testing Tool for Java: Benchmarking," In *International Workshop on Future Internet Testing*, pp. 101–110. Springer, 2014.

[19] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc. "Instance Generator and Problem Representation to Improve Object Oriented Code Coverage," *Transactions on Software Engineering*, pp. 294–313, 2015.

[20] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," Empir. Softw. Eng., vol. 11, no. 1, pp. 5–31, 2006.