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

Ryosuke Kurachi*, Eunjong Choi†, Dongsun Kim‡,Keichi Takahashi*, and Hajimu Iida*
*Nara Institute of Science and Technology, Japan, {kurachi.ryosuke.kp0, keichi}@is.naist.jp, iida@itc.naist.jp
†Kyoto Institute of Technology, Japan, echoi@kit.ac.jp
‡FuriosaAI, South Korea, darkrsw@furiosa.ai

***Abstract*—Automatically generated tests generally ignore the development process and intention behind the target code, and therefore generally tend to be less readable and maintainable. Reusing existing tests might solve this problem. To this end, we developed SuiteRec, a system that recommends high-quality reusable test suites based on code clone detection. Given a java Method, SuiteRec detects its code clones from OSS projects and then shows test suites of the cloned clones. It provides the ranking that shows the test suites based on the test smells and similarity between the input code and the cloned code. We evaluate SuiteRec with a human study of 10 student developers. The results indicate that SuiteRec successfully recommends high-quality reusable test suites.**

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

I. Introduction

In recent years, the requirements for software have become more sophisticated and diversified, while the demands from users for ensuring software quality and reducing costs have also increased. Among them, it is important to support software testing, which accounts for a large percentage of the total software development cost [**?**]. However, at present, most of unit tests are developed manually, and if you try to develop many tests, the cost will increase proportionally. Against this background, various automation technologies have been proposed to ensure software quality and achieve cost reduction [**?**], [**?**], [**?**], [**?**], [**?**].

For example, EvoSuite [**?**] is the most advanced tool for automatic unit test generation. EvoSuite statically analyzes the target code and expresses the program as symbolic values. Then, conditions that pass the control path of the target code are collected, and concrete values that satisfy the conditions are generated. By automatically generating tests, developers can save creation time and increase code coverage. However, the automatically generated test code is low readability and is not trusted by developers because it is not based on the development process and intention behind the target code, which makes later maintenance activities difficult [**?**], [**?**], [**?**]. Every time a test fails, the developer has to identify the cause of the failure in the production code or determine whether the test itself needs to be updated. Previous studies have reported that automatically generated tests are harder to read outweigh the time-savings gained by their automated generation, and render them more of a hindrance than a help for software maintenance [**?**].

In this paper, to solve this problem, we propose SuiteRec, a recommendation tool to find existing high-quality test codes from OSS projects.

The contributions from SuiteRec are shown below:

• **Test suite recommendation method.** We proposed test suites search method from OSS projects. The basic idea behind SuiteRec is that test codes can be reused between clone pairs. SuiteRec finds code clones of the input code from OSS projects and recommends test suites corresponding to the clones. The recommended test suites are sorted by their quality and presented to the developer. Furthermore, test smells, which indicate bad implementation of test codes, are shown for each test suite.

• **Quantitative and qualitative evaluation of SuiteRec.** We asked subjects to develop test code with and without SuiteRec, and compared the developed tests to evaluate how much test development can be supported. As a result, using SuiteRec is effective in increasing code coverage when developing test code for target code with many conditional branchesand test codes developed using SuiteRec have higher quality and less test smells. In addition, qualitative evaluation by questionnaire after the experiment showed that subjects feel that it is easier to develop test codes, and they are more confident in the resulting test codes when using SuiteRec.

II. BACKGROUND AND RELETED WORK

In the unit test execution task, the software is run, and it is confirmed whether the software behaves as expected in each test case. In order to reduce the cost of the test process, in test execution tasks, the use of automatic test execution tools such as JUnit is advancing in the industry for unit tests. However, test design tasks are still often performed manually, and the practical application and popularization of automation technology is expected.

Test cases developed by unit test design tasks consist of test procedures, test input values, and expected test results. The test input value is given to the software under test according to the test procedure, and the output result is compared with the expected test result. If they match, the test passes, otherwise it fails. In unit test design tasks, test input values are often developd using test case creation techniques such as equivalence partitioning and boundary analysis, but there are many variations to verify that the software works as required. You need to develop test input values for.

```java
public class ConvertString {
    public static String convertSnakeCase(String name) {
        if (name == null) throw new NullPointerException();
        String method = name;
        Pattern p = Pattern.compile("([A-Z])");
        for (;;) {
            Matcher m = p.matcher(name);
            if (!m.find()) break;
            method = m.replaceFirst("_" + m.group(1).toLowerCase());
        }
        return method.replaceFirst("^_", "");
    }
}
```

(a) Target Code fragment

```java
public class StringUtils {
    public String toSnakeCase(String text) {
        if (text == null) throw new NullPointerException();
        String snake = text;
        Pattern p = Pattern.compile("([A-Z])");
        for (;;) {
            Matcher m = p.matcher(snake);
            if (!m.find()) break;
            snake = m.replaceFirst("_" + m.group(1).toLowerCase());
        }
        return snake.replaceFirst("^_", "");
    }
}
```

(b) Similar Code fragment

```java
public class StringUtilsTest {
    @Test(expected = NullPointerException.class)
    public void expectedExceptionfor_null() throws Exception {
        StringUtils sut = new StringUtils();
        sut.toSnakeCase(null);
    }

    @Test
    public void returnSnakeCasefor_HelloWorld() throws Exception {
        StringUtils sut = new StringUtils();
        String expected = "hello_world";
        String actual = sut.toSnakeCase("HelloWorld");
        assertThat(actual, is(expected));
    }
    ...
}
```

(c) Test suite for similar code fragments

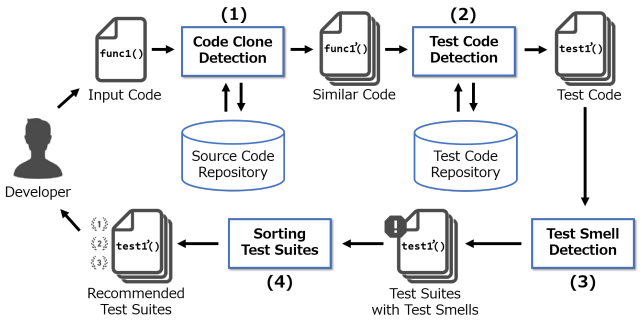

Fig. 1. Overview of SuiteRec

Test Smell. The importance of having well-designed test code was initially put forward by Beck [**?**]. Beck argued that test cases respecting good design principles are desirable since these test cases are easier to comprehend, maintain, and can be successfully exploited to diagnose problems in the production code. Inspired by these arguments, van Deursen et al. [**?**] coined the them test smells and defined the first catalog of 11 poor design choices to write tests, together with refactoring operations aimed at removing them. Such a catalog has been then extended more recently by practitioners, such as Meszaros [**?**] who defined 18 new test smells. As reported by recent studies, their presence might not only negatively affect the comprehension of test suites but can also lead to test cases being less effective in finding bugs in production code [**?**].

III. SuiteRec

SuiteRec detects cloned code of given input and then identifies test suites of that are correspond to the developer's input. Furthermore, test smells in test suites are detected, and test suites are ranked in descending order of quality.

Figure 1 shows the flow until test suites are recommended by SuiteRec . The recommendation method mainly consists of the following four steps.

(1) When SuiteRec receives a input code, it searches the corresponding similar code fragments from the Source Code Database using a existing clone detection tool.
(2) Detected similar code fragments, SuiteRec searches test suites corresponding to similar code fragments from the Test Code Database.
(3) SuiteRec detects test smells in the test suites collected by the previous step using a test smell detection tool.
(4) As the final step, SuiteRec ranks recommended test suites based on similarity and number of test smells.

A. Code Clone Detection

At first, SuiteRec detects code clones of the input code. To detect In this study, NICAD [**?**] was adopted as a code clone detection tool. NICAD detects clone pairs by converting the layout of code fragments and comparing code fragments line by line. By taking this approach, NICAD has realized clone pair detection with high accuracy and high recall. SuiteRec searches for projects that include input code by using NICAD, and outputs similar codes corresponding to the input codes in the detected clone pairs.

The Source Code Database in Figure 1 contains only target code fragments of the Github projects with test codes. Specifically, we selected projects that had test folders in the projects and adopted the JUnit testing framework.

NICAD has a project size limit that can be searched at once. In oerder 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 code fragments in real time. The detection setting is implemented in the SuiteRec 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 linked with test code

TABLE I
SUBJECT TEST SMELLS

Name	Description
Assetion Roulette	Occurs when a test method has multiple 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. Conditional code with in 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.

fragments. In this research, the following two steps are taken in order to precisely link the target code with test code fragments.

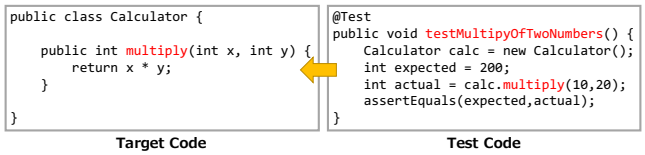

Fig. 2. Example of mapping test code to target code.

(1) Statically analyze the test code and obtain the method call of the target code.
(2) Divide the test method with a delimiter or capital letter and link it when the target method partially matches.

In the unit test, the target object is generated in the test code as shown in the figure 2, and it is executed by calling a method of the target code. Therefore, to link the target code to the test code, the test code in the Test Code Database is statically analyzed and the method call is obtained. However, multiple methods may be called within one 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 Database in Figure 1 stores test code corresponding to the production code in The Source Code Database. As a pre-processing, static analysis was performed on large-scale projects in advance, and information that linked target 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[1] was adopted as a test smell 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 to detect 6 test smells (TABLE I), which are important in considering the recommendation of test codes among 19 test smells that can be detected by tsDetect [**?**].

In addition, the test suites including the following four test smells that are not suitable as have been deleted from the Test Code Database in avance, so that it is not output as a recommended test suites.

- **Empty Test.** It occurs when a test method does not contain executable statements.
- **Ignored Test.** It means that test code has the @Ignore annotation and could not be executed.
- **Redundant Assertion.** It occurs when test methods contain assertion statements that are either always true or always false.
- **Unknown Test.** It indicates that a test method that does not contain a single assertion statement.

D. Test Suites Recommendation

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 projects.

As a result, there is 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 smells when the similarities are the same.

(1) **Input Code fragment.** The target code entered by the developer is displayed.
(2) **Similarity Code fragment.** 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.

[1]http://testsmells.github.io/

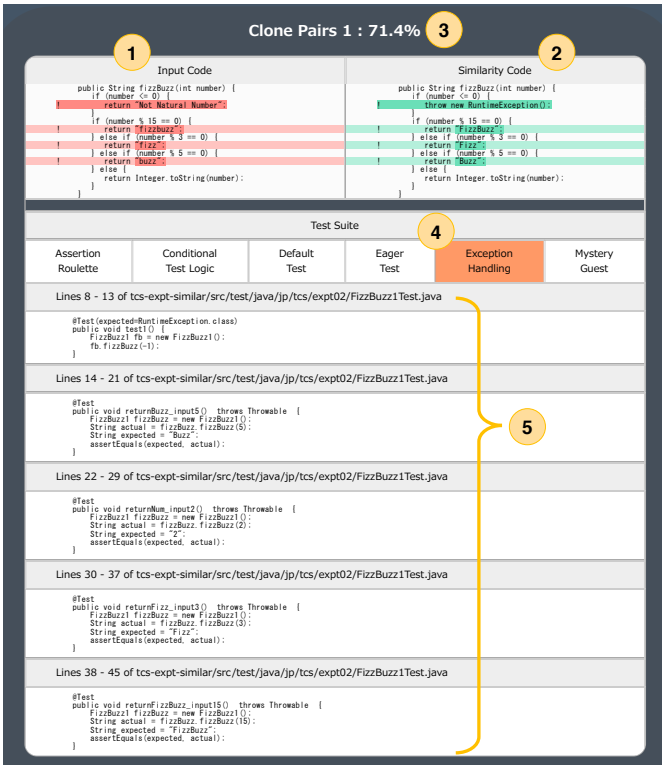

Fig. 3. Test suite recommended by SuiteRec.

(3) **Degree of similarity.** The similarity between the input code fragment and the similar code fragment 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, test smells are highlighted in orange, and the developer is presented with the presence of test smells.

(5) **Recommend Test Suites.** The recommended test suite is displayed. File paths are also displayed to indicate from which project the test code was referenced.

IV. Preliminary User Study

To quantitatively and qualitatively evaluate SuiteRec, we conducted a preliminary user study with 10 students developers, aiming at answer the following four research questions:

- **RQ1: Can SuiteRec support the development of test codes with high test coverage?** Test coverage is an important indicator of how well the software has been tested. We set up this RQ to confirm whether SuiteRec can contribute to increasing quality of the software.
- **RQ2: How well can SuiteRec reduce test suites generate time?** Generally, it takes much time to generate test suites from scratch. We set up this RQ to confirm whether SuiteRec can contribute to shorten the test suites generation time.
- **RQ3: How well SuiteRec support generate test codes with high quality?** We set up this RQ to confirm whether SuiteRec can contribute to generating high-quality test codes (i.e. with a fewer test smells).
- **RQ4: How effective SuiteRec is to support test suites generation tasks?** We set up this RQ to confirm how well can SuiteRec support developers in generating test suites.

For the preliminary user study, we recruited the students who have basic programming skills and software testing knowledge. The user study was conducted with 10 master course students who majored in information science. Among participants, more than 90% of participants had more than two years of programming experience, and more than 80% of participants had more than one year of Java language experience. All participants had a basic knowledge about software testing with lectures and other situations, and more than 80% had experience in developing unit tests.

The preliminary user study is comprised of two sessions, tutorial session and task session. In the tutorial session, we gave a quick lecture about basic knowledge about software testing and conducted an exercise of generating JUnit test suites to provide the deep understanding of test code generation. The tutorial session conducted during 30-minute.

In the task session, we assigned three tasks of creating JUnit test suites to participants with and without SuiteRec. We also requested participants to submit generated test suites. Hereafter, we call each of these assigned tasks as task1, task2 and task3. Among assigned tasks, task 3 was relatively difficult compared with task 1 and task 2. In each task, we provided a production code, targets of generating test suite. The number of conditional branch in the product code used in each task was 8, 16, and 24, respectively. We also gave a requirement specification written in natural languages, so that participants could readily the understand the specification of the production code for generating test suite. To mitigate the learning effect of SuiteRec, participants were assigned counterbalanced tasks by dividing into two group. In other words, tasks were conducted with SuiteRec for task 1, without SuiteRec for task 2, with SuiteRec for task3 in the one group. The tasks were conducted without SuiteRec for task 1, with SuiteRec for task 2, without SuiteRec for task3 for other group. Each task has a maximum of 25 minutes

The subjects were not allowed to refer to past answers. At the end, we conducted a survey to solicit feedback .

V. Results

This section presents the results of quantitative and qualitative evaluation of SuiteRec with 10 student developers, for each of the research questions.

A. RQ1: Can SuiteRec support the development of test codes 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[2], which is installed as a plug-in of the integrated development environment Eclipse[3]. Figures

[2]https://www.eclemma.org/
[3]https://www.eclipse.org/

4 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. 4, 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 developing test code for production code with many branches.

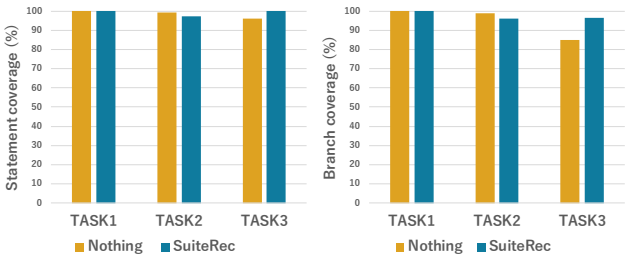

Fig. 4. Code coverage (C0,C1).

B. RQ2: Can SuiteRec reduce test codes development 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 task 1 and 3 than when it is not. This result can take time to read and understand multiple test suites recommended by SuiteRec. The subjects will not be able to reuse the recommended test codes without modification. It is necessary to edit test codes by looking at the difference between the input code and the detected similar codes.

For Task 2, it can be seen that the test creation time is shorter when SuiteRec is used. We investigated the submitted test codes and found that there was a lot of duplication of test cases when SuiteRec was not used even though there was no difference in coverage. This result suggests that the subjects may have wasted time developing many useless test codes.

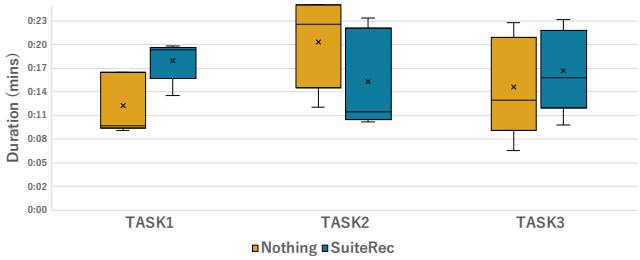

Fig. 5. Time taken to develop test codes.

C. RQ3: Can SuiteRec support the development of test codes with high quality?

Figure 6 shows the results of comparing the number of test smells in the submitted test suites with and without SuiteRec. For all tasks, the test codes developed using SuiteRec contains less test smell than if it were not used. This result suggests that the quality of the recommended test codes are high, and the subjects can develop the test codes while maintaining the quality by reusing them. 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.

As a whole, 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 [**?**].

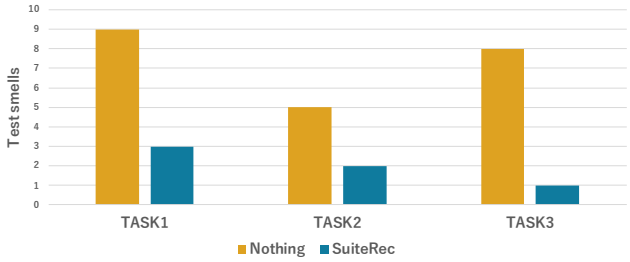

Fig. 6. Number of detected test smells.

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

Figure 7 summarizes answers to the survey questions. When developing test codes, subjects can easily feel test code creation using SuiteRec (Question 1). However, this result contrasts with the actual length of task end time (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 test suites and determine whether to reuse them. In addition, the test code cannot be applied without modification, and it is necessary to understand the difference between the input code and the detected similar codes and make appropriate modifications to the test codes. We speculate that when using SuiteRec, subjects may spend a lot of time on this part.

According to survey SuiteRec improvements, many responses said that it would be better to add functions that support test code editing such as a function that automatically edit classes and methods to names corresponding to the input code. Further improvements in SuiteRec show the potential for reducing the completion time of experimental tasks.

When using SuiteRec, the subjects are confident in the coverage of the test code developed by themselves (Question 2). On the other hand, 40% of subjects reported negative

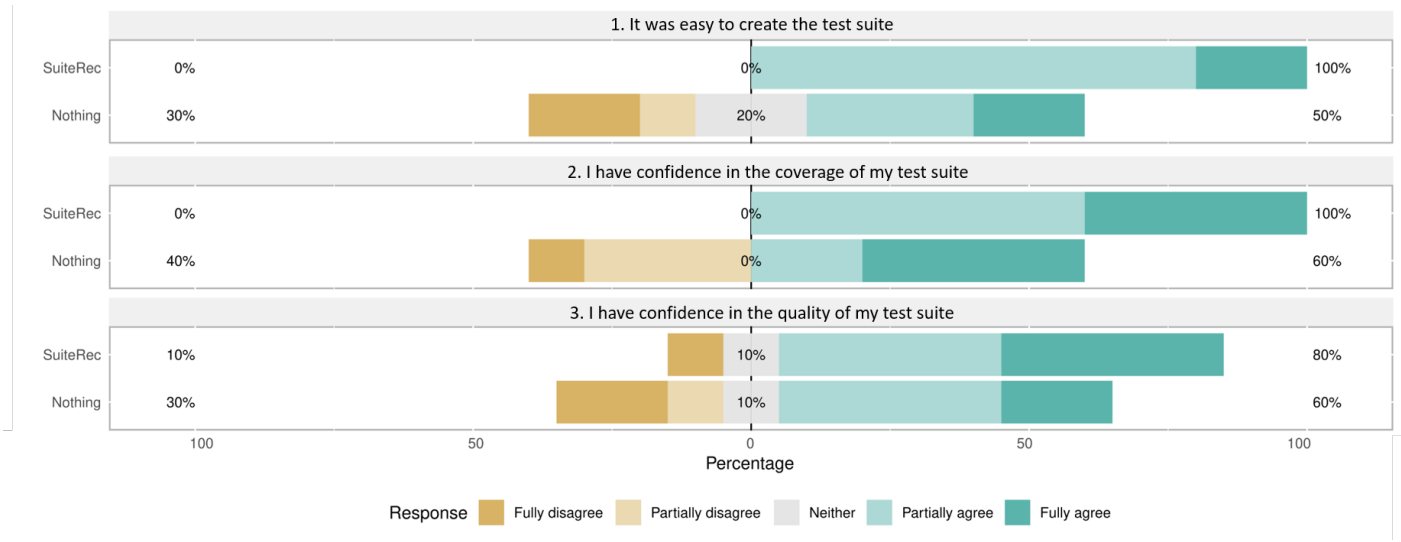

Fig. 7. Overview of the survey responses relating to test code creation tasks

responses when nothing was used. However, it is known that there is almost no difference in the coverage of the test code actually submitted (Figure 4). It is important to be confident in the coverage of the test codes they developed. One of the purposes of software testing is that developers are responsible for the code they develop and can provide software to users without anxiety.

When the test code was developed without using SuiteRec, 40% of the subjects were not confident in the quality of the test code they wrote (Question 3). It can be seen that the number of test smells in the actual submitted test codes are greater when SuiteRec is not used than when it is used (Figure 6). Developers unknowingly embed test smells that make later maintenance activities difficult. The use of SuiteRec reduces the number of test smells and brings confidence to the code they developed by giving developers an awareness of the quality of the test code.

On the other hand, even when SuiteRec is used, there are negative opinions about the quality of the test codes. This indicates that developers who are not familiar with test smells may become confused when they were presented with test smells.

VI. Related Work

Test Reuse for Clones. Zhang [a] et al. Proposed Grafter to transfer code between clone pairs, compare the test results before and after transplantation, and reuse the test based on that information. Soha [b] et al. Proposed Skipper that semi-automatically reuses and transforms the relevant part of the test suite corresponding to the code fragment when the developer reuses the code fragment. Skipper's approach is based on Gilligan [10], [34] and requires developers to define a detailed reuse plan to guide the conversion process. SuiteRec differs from these tools in two perspectives. First, SuiteRec searches for similar codes from projects on OSS. It is possible to find many test suites by searching in large projects. Second, SuiteRec only recommends test suites, leaving the detailed test reuse plan between clone pairs to the developer. Even if tests can be reused automatically, spreading low quality code makes maintenance difficult. SuiteRec presents test smells and gives developers confidence in the tests they developed by refactoring themselves.

VII. Discussion

A. Can SuiteRec support test code development?

When developing test codes for target codes (TASK1, TASK2) with a simple structure, it is an important result that there is no difference in coverage depending on whether SuiteRec is used (RQ1). Considering that you spend a lot of time developing test code when using SuiteRec, you can save time by developing test codes without using anything. On the other hand, when developing test codes for complex target codes with many branches (TASK3), using SuiteRec can increase the coverage by more than 10%. This result suggests that the recommended test suite provides useful information for developers to consider test items.

Responses to the free-form questions in the survey seem to corroborate this intuition. For example, one subject stated: *"I could notice test items that was overlooked by looking at the ."*

B. Can SuiteRec support the development of highly maintainable test codes?

RQ3 results showed that the quality of the test codes was high when SuiteRec was used. In the answer to free-form questions, there is a response that *"I understand the presented test smells and refactored it to eliminate test smells"*. Furthermore, there is a response that it was useful for improving readability, *"The test method name of the were useful when thinking about the method name"*. Existing research reports that automatically

generated test code is affected by test smells []. Mass generated test suites spread test smells in the project and have a significant impact on maintenance activities. this could end up being repaid during maintenance the cost that could be reduced by automatic generation. Determining whether this is the case requires further research and depends on many factors such as maintenance costs and project stability.

C. Threats to Validity

SuiteRec searches similar codes corresponding to the input code from OSS projects and recommends test suites of similar code. Therefore, the test suite cannot be recommended unless a similar code to the input code is found. Even if similar codes can be detected, if the degree of similarity is low, the recommended test suite may not be useful. NICAD detects type 2 and 3 code clones. Code clones (type 3) with insertion and deletion of statements may behave differently. In such cases, it is difficult to reuse tests. Therefore, the test suites that can be recommended may be limited to a narrow range.

VIII. Conclusion and Future Work

SuiteRec is a recommendation tool that find existing high-quality test codes from OSS projects. SuiteRec finds code clones of the input code from OSS projects and recommends test suites corresponding to the clones. The recommended test suites are sorted by their quality and presented to the developer. Furthermore, test smells, which indicate bad implementation of test codes, are shown for each test suite.

In the evaluation, we asked subjects to develop test codes with and without using SuiteRec and compared the developed test codes. We show that (1) SuiteRec improves code coverage when developing test codes for target codes with many conditional branches, (2) test codes developed using SuiteRec have higher quality and less test smells, and (3) developers feel that it is easier to develop test codes, and they are more confident in the resulting test codes when using SuiteRec.

The following tasks are planned for future work:

- SuiteRec needs to be improved for more practical use. Specifically, we plan to add a test code automatic editing function and refactoring function for test smells.
- Further experiments with more subjects are needed to measure the significance of SuiteRec. In addition, To eliminate the influence of experience, it is necessary to conduct experiments not only on students but also on engineers with extensive test development experience.
- To investigate whether SuiteRec can be recommended in the order of preference required by developers, we plan to conduct a validity evaluation on the ranking of the test suites by subjects.