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.nais
†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 a tool named SuiteRec, which recommends high-quality test suites based on code clone detection. SuiteRec detects code clones from the input code and then suggests test suites of cloned code. It ranks the test suites based on the number of test smells and similarity between the input code and the corresponding code. We evaluated SuiteRec with a human study of 10 student developers, and the results showed that SuiteRec support developers to generate high-quality 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 [21]. However, at present, most of unit tests are created manually, and if you try to create many tests, the cost will increase proportionally. Against this background, various automation technologies have been proposed to ensure software quality and achieve cost reduction [3], [17], [18], [19], [20].

For example, EvoSuite [3] 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 process and intention of creating the target code, which makes later maintenance activities difficult [14], [15], [16]. 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 [1].

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 created 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 created 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 create 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

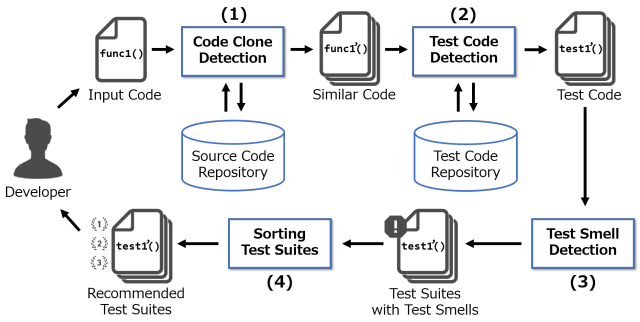

Fig. 1. Overview of SuiteRec.

Test Smell. The importance of having well-designed test code was initially put forward by Beck [4]. 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. [7] 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 [6] 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 [8].

III. SuiteRec

SuiteRec takes a code fragment from the developer and searches in exsiting OSS projects to identify test suites 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 4 steps.

(1) When SuiteRec receives a input code, it searches the source code repository for the corresponding similar code fragments using a existing clone detection tool.

(2) Detected similar code fragments, SuiteRec searches the test code repository for test suites corresponding to similar code fragments.

(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

In this study, NICAD [2] 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. NICAD searches a large number of projects in GitHub for similar code fragments corresponding to the input code.

The Source Code Repository 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 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 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 fragments. In this research, the following 2 steps are taken in order to precisely link the target code with test code fragments.

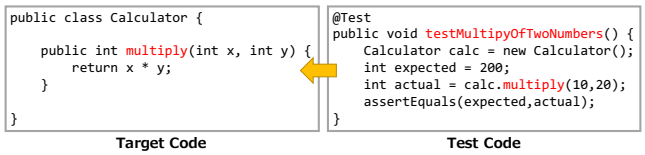

Fig. 2. Example of mapping test code to target code.

(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 test target code to the test code, the test code in the Test Code Repository is statically analyzed and the method call is obtained. 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 [22]. 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 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 [10] 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 the following 6 test smells, which are important in considering the recommendation of test codes among 19 test smells that can be detected by tsDetect [9].

In addition, the test suites including the following 4 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 suites.

- **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.
- **Unknown 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 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.

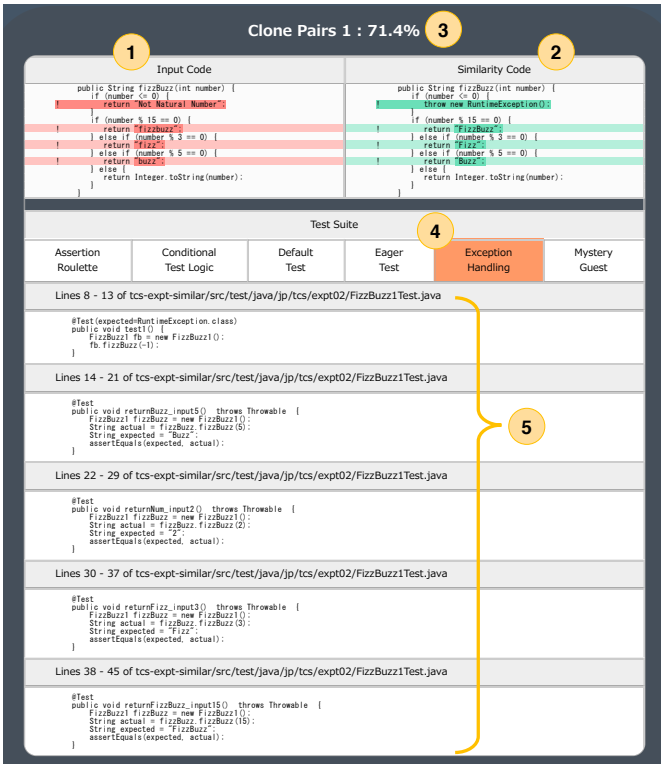

Fig. 3. Test suite recommended by SuiteRec.

(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.

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 thefailure 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.

(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. EXPERIMENTAL SETUP

In this section, we will conduct experiments with the subjects to evaluate SuiteRec quantitatively and qualitatively. The subjects will be asked to create test suites 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 statement 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 the subjects 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 subjects 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 the subjects had basic knowledge about software testing in lectures and other situations, 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, specifications written in natural language were 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 subjects fully understood the given experimental task and had sufficient work time.

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 suites for the three production codes for the actual experiment.

Ask the subjects to judge the end of the experimental task. Specifically, the 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. Finally, after the completion of the experimental task, subjects were asked to answer a questionnaire on test code development.

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 [11], which is installed as a plug-in of the integrated development environment Eclipse [12]. Figures 5 and 6 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. 6, 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.

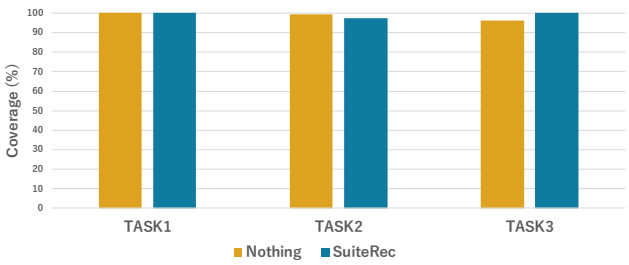

Fig. 4. Statement coverage (C0).

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

Figure 7 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. The subjects will not be able

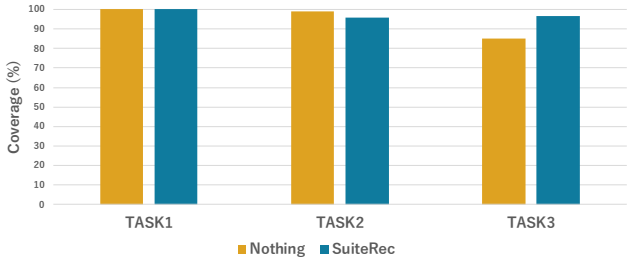

Fig. 5. Branch coverage (C1).

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 statement 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 (test 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.

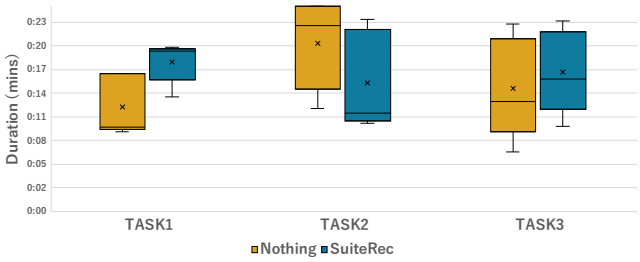

Fig. 6. Time taken to create test code.

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

Figure 8 shows the results of comparing the number of test smells in the submitted test suites 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 subjects can create the test code while maintaining the 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 [9].

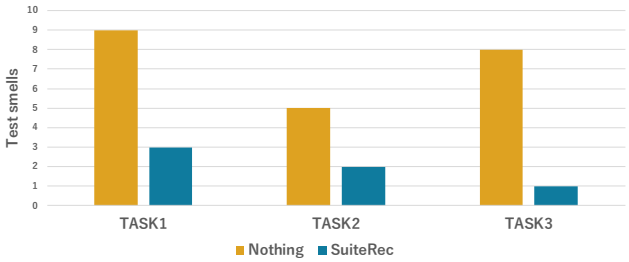

Fig. 7. Number of detected test smells.

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

Figure 9 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 we received. 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 created by themselves (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 5,6). 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 8). 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.

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] Beck. *Test Driven Development: By Example.* Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[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.

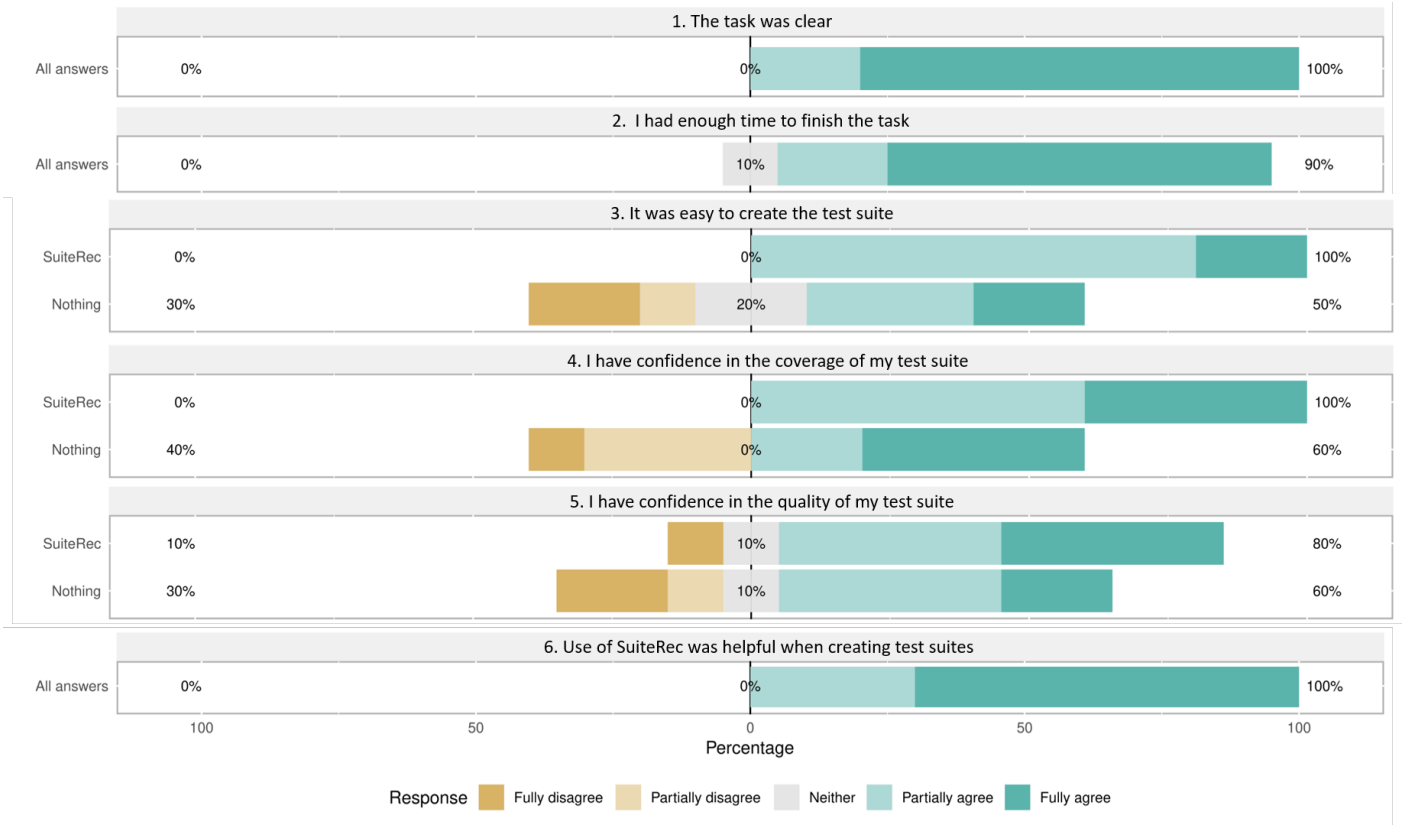

Fig. 8. Overview of the survey responses relating to test code creation tasks

[9] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and 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] https://www.eclipse.org/

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] F. Appel. Testing with JUnit. Packt, 2015.