UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

WESLEY BRENNO RODRIGUES HERCULANO

GENERATED TESTS IN THE CONTEXT OF MAINTENANCE TASKS: A SERIES OF EMPIRICAL STUDIES

CAMPINA GRANDE – PB
2022

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Generated Tests in the Context of Maintenance Tasks: a Series of Empirical Studies

Wesley Brenno Rodrigues Herculano

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Nome dos Orientadores

Everton L. G. Alves

Melina Mongiovi

Campina Grande, Paraíba, Brasil

©Wesley Brenno Rodrigues Herculano, 04/10/2022

H539g Herculano, Wesley Brenno Rodrigues.

Generated tests in the context of maintenance tasks: a series of empirical studies / Wesley Brenno Rodrigues Herculano. – Campina Grande, 2022.

87 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2022.

"Orientação: Prof. Dr. Everton L. G. Alves, Profa. Dra. Melina Mongiovi".

Referências.

1. Software Engineering. 2. Generated Tests. 3. Maintenance. 4. Randoop. 5. Evosuite. 6. Test Smells. 7. Refactoring. I. Alves, Everton L. G. II. Mongiovi, Melina. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

WESLEY BRENNO RODRIGUES HERCULANO

GENERATED TESTS IN THE CONTEXT OF MAINTENANCE TASKS: A SERIES OF EMPIRICAL STUDIES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 04/10/2022

Prof. Dr. EVERTON LEANDRO GALDINO ALVES, UFCG, Orientador

Prof. Dra. MELINA MONGIOVI BRITO LIRA, UFCG, Orientadora

Prof. Dra. PATRICIA DUARTE DE LIMA MACHADO, UFCG, Examinadora Interna

Prof. Dra. ROBERTA DE SOUZA COELHO, UFRN, Examinadora Externa

Documento assinado eletronicamente por **EVERTON LEANDRO GALDINO ALVES**, **PROFESSOR 3 GRAU**, em 04/10/2022, às 13:57, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Roberta de Souza Coelho**, **Usuário Externo**, em 04/10/2022, às 15:38, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO**, **COORDENADOR DE POS-GRADUACAO**, em 04/10/2022, às 17:17, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **PATRICIA DUARTE DE LIMA MACHADO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 04/11/2022, às 16:56, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **2746353** e o código CRC **5AD8CA3A**.

Referência: Processo nº 23096.064751/2022-51 SEI nº 2746353

Resumo

As atividades de manutenção geralmente consomem tempo e são difíceis de gerenciar. Para lidar com isso, os desenvolvedores geralmente usam casos de teste que falham para orientar seus esforços de manutenção. Portanto, possuir bons casos de teste é essencial para o sucesso da manutenção. Testes gerados automaticamente podem economizar tempo e obter uma maior cobertura de código. No entanto, muitas vezes não refletem cenários realistas e incluem test smells. Além disso, não é clara a eficácia destes testes ao orientar as atividades de manutenção, nem se os desenvolvedores os aceitam totalmente. Neste trabalho, apresentamos uma série de estudos empíricos que avaliam se testes gerados automaticamente podem dar suporte aos desenvolvedores na manutenção do código. Primeiro, realizamos um estudo com 20 desenvolvedores para comparar como eles executam atividades de manutenção com testes gerados automaticamente (Evosuite ou Randoop) e testes escritos manualmente, obtendo resultados que mostram que testes gerados automaticamente podem ser de grande ajuda para identificar falhas durante a manutenção, com os desenvolvedores sendo mais precisos na identificação de atividades de manutenção ao usar os testes gerados pelo Evosuite e igualmente eficazes nas correções de bugs usando testes manuais e testes gerados pelo Evosuite e Randoop. Em seguida, aplicamos um questionário com 82 desenvolvedores para avaliar a percepção sobre o uso de testes do Randoop (ferramenta que, de maneira geral, apresentou o pior desempenho no primeiro estudo) refatorados removendo 3 tipos de testes smells, onde os resultados demonstram que os desenvolvedores preferiram testes do Randoop refatorados aos originais. Por fim, realizamos um terceiro estudo empírico com 24 desenvolvedores focado em avaliar o impacto dos Randoop refatorados em atividades de manutenção, com resultados evidenciando que as refatorações aplicadas não melhoraram o desempenho dos desenvolvedores na detecção de falhas. Por outro lado, os desenvolvedores foram mais eficazes na correção das faltas usando testes Randoop refatorados.

Palavras-chave: testes gerados, manutenção, randoop, evosuite, *test smells*, refatoração.

Abstract

Maintenance tasks are often time-consuming and hard to manage. To cope with that, developers often use failing test cases to guide their maintenance efforts. Therefore, working with good test cases is essential to the success of maintenance. Automatically generated tests can save time and lead to higher code coverage. However, they often do not reflect realistic scenarios and include test smells. Moreover, it is not clear whether generated tests can be effective when guiding maintenance tasks, nor if developers fully accept them. In this work, we present a series of empirical studies that evaluate if automatically generated tests can support developers when maintaining code. First, we ran an empirical study with 20 real developers to compare how they perform maintenance tasks with automatically generated (Evosuite or Randoop) and manually-written tests. Our results showed that automatically generated tests can be a great help for identifying faults during maintenance. Developers were more accurate at identifying maintenance tasks when using Evosuite tests and equally effective to create bug fixes when using manually written, Evosuite, and Randoop. Then, we applied a survey with 82 developers to assess developers' perceptions of the use of Randoop tests (that presented the worst performance in the first study) refactored by removing 3 kinds of test smells. Results of this investigation showed that developers preferred refactored Randoop tests to the original ones. Finally, a third empirical study with 24 developers focused on evaluating the impact of these refactored Randoop tests on maintenance performance. We found that the refactorings applied did not improve their performance in detecting the faults. On the other hand, developers were more effective in fixing the faults using refactored Randoop tests.

Keywords: generated tests, maintenance, randoop, evosuite, test smells, refactoring.

Agradecimentos

A Deus, toda honra e toda a glória, é d'Ele esta vitória alcançada em minha vida.

À minha família, meus pais Herculano e Rosa, a minha base de tudo, que não mediram esforços para me proporcionar a melhor educação possível, e criar todas as condições necessárias para que eu pudesse alcançar meus objetivos. Aos meus irmãos, Douglas e Hércules, por todo o apoio e incentivo nos estudos durante toda minha caminhada.

À minha esposa, Eloísa, por todo companheirismo e incentivo, por aguentar meus estresses diários, e em todos os momentos de ansiedade, cansaço e preocupações, ser a minha calmaria.

Aos professores Everton e Melina, por serem os melhores orientadores possíveis, sempre com muita dedicação e responsabilidade com suas atividades, por todo conhecimento repassado, todo o apoio e paciência, e por me fazerem desistir de desistir algumas vezes.

Aos amigos do Ap. 15, que me acompanharam durante toda a graduação e no início do mestrado, dividindo além da moradia, as alegrias e perrengues cotidianos.

Aos meus amigos e colegas do Virtus, que de forma direta ou indireta contribuíram com este trabalho.

A todos os colaboradores do Computação@UFCG e professores que fizeram parte da minha formação, desde a infância até hoje, contribuíndo no "assentamento de tijolos" na edificação do meu conhecimento.

A todos voluntários, pela solicitude e tempo despendido para a participação nos experimentos descritos neste trabalho.

E ao meu filho, José Benício, que chegou na reta final deste curso, mas que desde então, tem resignificado tudo em minha vida.

Contents

1 Introduction **1**

1.1 Context 1

1.1.1 Motivational Example 2

1.2 The Problem 3

1.3 Objectives 5

1.4 Results and Implications 6

1.5 Relevance 8

1.6 Work organization 8

2 Background **10**

2.1 Test Generation Tools 10

2.1.1 Randoop 10

2.1.2 EvoSuite 11

2.2 Test smells 11

2.3 Refactoring Test Cases 12

2.4 Software Maintenance 13

2.5 Final Considerations 14

3 A Study On The Use Of Generated Tests To Guide Maintenance Tasks **15**

3.1 Motivation 15

3.2 Design and Research Questions 16

3.2.1 Participants Selection and Demographics 17

3.2.2 Study Objects 17

3.2.3 Study Procedure 20

3.3 Results and Discussion 24
3.3.1 RQ1: Do generated tests influence the effectiveness of developers on determining the source of a problem? 25
3.3.2 RQ2: Are generated tests effective to help to find proper fixes? . . . 26
3.3.3 RQ3: Does it take longer to execute maintenance tasks when using generated tests instead of manually written ones? 29
3.3.4 RQ4: What is the developers' perception of using generated tests when performing maintenance tasks? 31
3.3.5 Analysis by Roles 33
3.3.6 Diverging Results 34
3.4 Final Considerations 34

4 A Survey to Evaluate Developers Perspectives on Refactored Tests 36
4.1 Motivation 36
4.2 Design and Research Questions 36
4.2.1 Methodology 37
4.2.2 Study Objects 38
4.3 Results and Discussion 42
4.3.1 Participants Demographics 42
4.3.2 RQ5: What is the developers' perception concerning the names of Randoop test cases? 42
4.3.3 RQ6: Do developers prefer the original Randoop tests or the refactored ones? 46
4.3.4 Analysis by Roles 47
4.4 Final Considerations 48

5 An Study on the Use of Refactored Generated Tests to Guide Maintenance Tasks 49
5.1 Motivation 49
5.2 Design and Research Questions 50
5.2.1 Participants Selection and Demographics 50
5.2.2 Study Objects 51
5.2.3 Study Procedure 52

5.2.4 Follow-up Interviews 53
5.3 Results and Discussion 53
5.3.1 RQ7: Do refactored Randoop tests improve developers' effectiveness in determining the source of an issue? 54
5.3.2 RQ8: Do refactored Randoop tests improve developers' effectiveness in performing proper fixes? 56
5.3.3 RQ9: Do refactored Randoop tests improve the developers' performance to execute maintenance tasks? 56
5.3.4 RQ10: What is the developers' perception about using refactored Randoop tests in maintenance tasks? 58
5.3.5 Analysis by Roles 61
5.3.6 Follow-up Interviews 63
5.4 Final Considerations 64

6 Threats to validity **65**

7 Related work **67**
7.1 Methodology 67
7.2 Comparing Manual and Generated Test Cases 68
7.3 Test Smells on Generated Tests 69
7.4 Test Code Improvements 69
7.5 Final Considerations 71

8 Conclusions and Future Works **72**

A Experiment procedure **82**

B Extra analysis **83**

List of Symbols

API - *Application Programming Interface*

AST - *Abstract Syntax Tree*

CUT - *Class Under Test*

IDE - *Integrated Development Environment*

QA - *Quality Assurance*

List of Figures

1.1 Test cases for the `ComparatorChain` class. 4

1.2 Example of a codefix for the `ComparatorChain` class. The `for` loop in the `compare` method iterates over all elements except the last one due to a wrong stop condition. 5

3.1 Study participantsâ background information. 18

3.2 Faulty version and possible fix for the `ListPopulation` class. 20

3.3 Example of manually written faulty test case for the `FixedOrderComparator` class. 21

3.4 Example of Randoop generated faulty test case for the `FixedOrderComparator` class. 22

3.5 Overview of our study procedure for each participant. 24

3.6 Protocol for defining correct code and test fixes. 25

3.7 The time developers spent to identify their maintenance tasks grouped by task type and class. 30

3.8 The time developers spent to fix their maintenance tasks grouped by task type and class. 30

3.9 Overview of the survey responses relating to (a) *codefix* and (b) *testfix* maintenance tasks. 32

4.1 Examples of questions from the second section 38

4.2 Examples of questions from the third section. 39

4.3 Renamed Randoop tests for `ListPopulation` class. 40

4.4 Original and split Randoop tests for `FixedOrderComparator` class. . . 41

4.5 Participants' background information. 43

4.6 Overview of answers about the agreement with automatically generated test names. 44
4.7 Overview of answers about more appropriate test names. 45
4.8 Answers about the test that should exercise a CUT. 45
4.9 Answers about the most readable test code. 47
4.10 Answers about the most appropriate test to include into a test suite. 48

5.1 Study participants' background information. 51
5.2 The time developers spent to identify their maintenance tasks grouped by task type and class. 60
5.3 The time developers spent to fix their maintenance tasks grouped by task type and class. 61
5.4 Overview of the questionnaire responses relating to (a) *codefix* and (b) *testfix* maintenance tasks. 62

A.1 Detailed procedure overview of the studies from chapters 3 and 5, considering a single participant 82

B.1 Overview of survey answers about the agreement with automatically generated test names, considering only software engineers and quality analysts participants. 83
B.2 Overview of survey answers about more appropriate test names, considering only software engineers and quality analysts participants. 84
B.3 Survey answers about the test that should exercise a CUT, considering only software engineers and quality analysts participants. 84
B.4 Survey answers about the most readable test code, considering only software engineers and quality analysts participants 85
B.5 Survey answers about the most appropriate test to include in a test suite, considering only software engineers and quality analysts participants 85
B.6 The time developers spent to identify their maintenance tasks grouped by task type and class, considering only software engineers and quality analysts participants. 86

B.7 The time developers spent to fix their maintenance tasks grouped by task type and class, considering only software engineers and quality analysts participants. 87

List of Tables

3.1 Information about the selected classes. Branch coverage values refer to the test suite used to evaluate the codefixes performed by the participants. . . . 19
3.2 Comparison of correct decisions given Manual, Evosuite or Randoop tests. . 27
3.3 Statistical analysis and ranking considering correct decisions for maintenance tasks. 27
3.4 Comparison of correct fixes using Manual, Evosuite or Randoop tests. . . . 28
3.5 Statistical analysis and ranking considering correct fixes for maintenance tasks. 28

5.1 Comparison of correct decisions given Randoop, Randoop split or Randoop split-renamed tests. 55
5.2 Statistical analysis and ranking considering correct decisions for maintenance tasks. 55
5.3 Comparison of correct fixes using Randoop (original), Split or Split-renamed Randoop tests. 57
5.4 Statistical analysis and ranking considering correct fixes for maintenance tasks. 57
5.5 Statistical analysis and ranking considering identification time for maintenance tasks. 59
5.6 Statistical analysis and ranking considering correction time for maintenance tasks. 59

B.1 Comparison of correct decisions given Randoop, Randoop split or Randoop split-renamed tests, considering only software engineers and quality analysts participants. 86

B.2 Comparison of correct fixes using Randoop (original), Split or Split-renamed Randoop tests, considering only software engineers and quality analysts participants. 86

Chapter 1

Introduction

In this chapter, we present an overview of our work. In section 1.1, we present the context in which this work is inserted. In section 1.2, we define and detail the problem addressed. The specific objectives of our work are listed in section 1.3. In section 1.4, we summarize the results obtained and their implications. In section 1.5, we present and describe the main contributions of our work. In section 1.6, we describe the structure of the document remainder.

1.1 Context

A software must be predictable and consistent, offering no surprise to the user. In this context, testing activities are important to ensure the quality of the software under development, to assess whether the program works according to its specification, and to reveal in advance as many faults as possible with the least effort [44].

Software tests can be found in four general levels: unit testing, integration testing, system testing, and acceptance testing [68]. Unit testing is the test of individual code units, or groups of related units [13]. However, building a good unit test suite is both difficult and time-consuming [17].

To cope with this issue, test generation tools have been proposed to create tests from scratch. These tools use generation strategies based on genetic algorithms, search-based algorithms, mutation-based assertion generation, or feedback-directed random generation (e.g., [39; 49; 24; 22]). From this wide range, we can highlight two of the most well-known test generation tools: Evosuite [22] and Randoop [47]. Both of them have been used as a baseline, and/or won awards, in the SBST Java Unit Testing Tool Contest [7; 34; 59; 14].

Unit testing plays an important role during maintenance tasks [8]. It often works as a safety net to avoid fault introduction when updating a code, and as a great help for bug identification and fixing, [45]. In addition to detecting and helping fix faults, good unit tests should be easily updated after source code changes [30], therefore, avoiding possible false alarms (aka false positives), i.e., when a test fails but production code is correct; and silent horror (aka false negatives), i.e., when no warning is issued, even though there are bugs in the production code [15]. These characteristics are not commonly found in automatically generated tests, which may prevent their use in practice.

1.1.1 Motivational Example

Here, we present code snippets from the `ComparatorChain`[1] class that was extracted from the Apache Commons project[2], and three test cases (manually written, generated by Evosuite, and generated by Randoop). The `ComparatorChain` class includes a fault. The `for` loop in the `compare` method iterates over all elements except the last one due to a wrong stop condition. Figure 1.2 illustrates a possible codefix for this fault. The test cases presented in Figures 1.1(a), 1.1(b), and 1.1(c) (manually written, Randoop, and Evosuite test) fail when we run them to test the faulty code of the `ComparatorChain` class.

The manually written test adds a new comparator in the `chain` object, from the `ComparatorChain` class, and tests it using some asserts. The Randoop test creates empty comparators and checks their sizes. The Evosuite test creates an empty comparator and expects a `NullPointerException`. Although the test cases exercise the same method, they have different testing purposes. All of them fail due to faulty code.

As discussed before, developers can save time by automatically generating test cases using a tool. However, a question that remains is whether automatically generated tests make maintenance tasks (codefixes and testfixes) harder. We asked seven developers to find the fault in the code of Figure 1.2. Four of them used the manually-written test case (Figure

[1] https://github.com/WesleyBrenno/generated-tests-in-the-context-of-maintenance-tasks-a-series-of-empirical-studies/blob/main/study-with-generated-tests/golden/golden_code/ComparatorChain.java

[2] https://commons.apache.org/

1.1(a)) but one did not find the fault. Three developers used the generated test (Figure 1.1(c)) and all of them found the fault. This example may suggest the need for further evaluation of this matter. In this work, we perform a series of empirical studies to evaluate how developers deal with maintenance tasks using variations of automatically generated tests.

1.2 The Problem

Software maintenance represents up to 60% of a project's budget [11; 28]. Previous work have evaluated automatically generated tests concerning code coverage, faults identification, and time-consumption [24; 22; 47]. However, to the best of our knowledge, there are only a few studies designed for evaluating whether automatically generated tests are maintainable and can properly support developers in software maintenance tasks.

Shamshiri et al. [61] ran an empirical study where participants faced maintenance tasks with the help of a failing test case. The failing test could be manually written or generated by Evosuite. The participants were asked to identify and fix the cause of the failure, which could be related to implementation or a specific unit test. They were more efficient (took less time) at maintenance tasks when using manually written failing tests but equally effective (produced correct fixes equally) with manually written and Evosuite generated tests. However, this study was conducted only in an academic scenario with undergraduate and graduate students and evaluated a single test generating tool.

Although automated test generation tools have gained notoriety recently as they are able to assist in discovering real faults in code [62] [55], can be a great alternative to reducing the costs of creating test suites and lead to higher coverage levels [22], the fact of those tests are often not close to realistic scenarios, are less readable [53], and have a high incidence of test smells [69] make it hard for developers to understand them. Figure 1.1 (b) and (c) illustrates tests generated by Randoop and EvoSuite, respectively. In those tests, we can find some issues, such as verbose code (Figure 1.1(b) lines 2 and 4), assertions that are not easy to read (Figure 1.1(b) lines 6 and 7), the lack of documentation, and non descriptive names for variables and test methods (Figure 1.1(b) lines 1 to 5, and Figure 1.1(c) lines 1 and 2). These issues may prevent developers from using the tests to locate/fix faults and/or to maintain them, which can lead to an increase in time and difficulties in software maintenance

```
public void test() {
    final ComparatorChain <Integer> chain = new ComparatorChain<>();
    chain.addComparator(new Comparator <Integer>() {
    @Override
    public int compare(final Integer a,final Integer b){
         ...
    }
    assertTrue(chain.size() == 1);
    assertTrue(chain.compare(Integer.valueOf(4),Integer.valueOf(5)) > 0);
    ...
}
```

(a) Test manually written for the `ComparatorChain` class.

```
public void test() throws Throwable {
        collections.comparators.ComparatorChain<java.lang.Comparable<java.lang.String>>
    strComparableComparatorChain0 = new collections.comparators.ComparatorChain<java.
    lang.Comparable<java.lang.String>>();
        int int1 = strComparableComparatorChain0.size();
        collections.comparators.ComparatorChain<java.lang.Comparable<java.lang.String>>
    strComparableComparatorChain3 = new collections.comparators.ComparatorChain<java.
    lang.Comparable<java.lang.String>>();
        int int4 = strComparableComparatorChain3.size();
        org.junit.Assert.assertTrue("'" + int1 + "' != '" + 0 + "'", int1 == 0);
        org.junit.Assert.assertTrue("'" + int4 + "' != '" + 0 + "'", int4 == 0);
    }
```

(b) Test generated by Randoop tool for the `ComparatorChain` class.

```
public void test() {
    ComparatorChain<String> comparatorChain0 = new ComparatorChain<String>(( Comparator<
    String>)null,true);
    // Undeclared exception!
    try{
        comparatorChain0.compare("S6jQ9HA[==\"e","T <]IB");
        fail("Expecting exception: NullPointerException");
    }catch(NullPointerException e){
    // no message in exception (getMessage() returned null)
    }
}
```

(c) Test generated by Evosuite tool for the `ComparatorChain` class.

Figure 1.1: Test cases for the `ComparatorChain` class.

```

/**
 * A ComparatorChain is a Comparator that wraps one or more Comparators in sequence.
... **/
public class ComparatorChain<E> implements Comparator<E>, Serializable {
    ...

    /**
     * Perform comparisons on the Objects as per Comparator.compare(o1,o2).
     ...**/
    public int compare (final E o1,final E o2) {
- for (int comparatorIndex=0; comparatorIndex < comparatorChain.size()-1; ++
    comparatorIndex) {
+ for (int comparatorIndex=0; comparatorIndex < comparatorChain.size(); ++
    comparatorIndex) {
        ...
    }
    ...
}
```

Figure 1.2: Example of a codefix for the `ComparatorChain` class. The `for` loop in the `compare` method iterates over all elements except the last one due to a wrong stop condition.

activities. Some approaches have been proposed to improve the test readability [26; 2; 50], as well as to detect and/or remove some types of test smells [72; 36]. However, as far as we know, there are only a few studies designed to assess whether automatically generated tests are maintainable and if they can support developers in software maintenance tasks, and how to improve them.

1.3 Objectives

The main goal of this work is to evaluate and improve the way developers deal with corrective maintenance tasks using automatically generated tests. We can break this goal into four specifics ones:

- Conduct an empirical study to analyze how developers perform maintenance tasks with automatically generated (Evosuite or Randoop) and manually-written tests;
- Adapt well-known refactoring strategies to reduce test smells in Randoop test cases,

with the objective of improving them;

- Evaluate how developers perceive and perform maintenance tasks with improved generated tests.

1.4 Results and Implications

In this work, we report a series of empirical studies ran to evaluate how developers deal with maintenance tasks using automatically generated tests. First, we ran a study with 20 developers from different companies and ask them to identify and fix the cause of a test failure. The fault could be either in the production code or the test code. We used real test failures produced by developers whilst performing implementation tasks, and compared manually written tests to tests generated automatically by EvoSuite and Randoop. For this study, we reused most of the design and artifacts from Shamshiri et al.'s work [61], but we applied a more realistic scenario (real developers) and introduced an extra test generation tool (Randoop).

From this study we yielded the following main results:

- Developers were more effective (i.e., had a better hit rate) at identifying maintenance tasks when using Evosuite tests, while they were equally effective when using manually written and Randoop tests;
- Developers were similarly effective at producing bug fixes using the three strategies (manually written, Evosuite, and Randoop);
- Developers were similarly efficient (i.e., spent the same time) at executing maintenance tasks using the three strategies (manually written, Evosuite, and Randoop);
- Developers found generated tests hard to read, specially Randoop's. The Evosuite test case structure was more appreciated but also requires improvements to increase readability and comprehension aspects.

These results reflect that, although easier to read, a manual test may not necessarily be better than generated tests to help locate and/or understand code faults. In this sense, generated tests might be a good option. Moreover, there is a need for improvements in automatically generated tests when used for maintenance purposes, especially Randoop tests, which performed worse in our first study.

Test smells may compromise test code comprehension, readability, and maintenance [27; 29]. Generated test cases can include a number of test smells, such as non-descriptive names, assertion roulette, duplicate assert, eager test, lazy test, and magic number test [69]. All of those can often be found in Randoop tests (Figure 4.4). This observation motivated us to perform a whole new study to evaluate the impact of well-known refactorings (Extract Method and Rename Method) on the quality of Randoop tests when applied in an automatic way. For that, we first conducted a survey with 82 software professionals, and ask their perceptions when comparing original Randoop tests to their automatically refactored versions.

From this survey we found that:

- Although we cannot say that developers preferred automatically generated test names over manually written ones, the automatic renaming of Randoop tests was well-received;
- Developers preferred refactored Randoop tests over original ones. However, in order to fully accepted them, they indicated the need for extra refactorings, such as variables renaming, and extract method.

The results from this survey motivated us to access the practical impact of the use of refactored tests on the performance of maintenance tasks. For this, we performed a third study where we replicated the first one but focused on different versions of Randoop tests (original and refactored ones). We can summarize the results of this investigation as follows:

- The refactorings did not improve, nor worsen, the performance of developers determining the root cause of the faults;
- Developers were more effective in performing proper fixes when guided by refactored Randoop tests;
- When guided by refactored Randoop tests, developers took less time fixing the faults, but the same was not observed for fault identification;

- Refactored Randoop tests contributed to a better understanding of the class under testing (CUT) and facilitated the identification of maintenance activities and code fixes. However, developers felt more confident about their test fixes, when using original Randoop tests.

1.5 Relevance

Our work is based on a series of empirical studies. We believe that those studies may provide valuable insights for developers and future research.

In our work, we compared different generated tests (Evosuite, Randoop, and Randoop refactored) regarding their impact on software maintenance. Our results showed that they can be a great help to support developers in this activity. This information can help developers decide which and how to adopt the use of generated tests in their projects. Moreover, this study may inspire other studies about the impact of the use of tests generated in software development.

Our work also demonstrates that well-known refactoring techniques can be applied to reduce test smells occurrences in generated tests, and how they can be used to improve the quality and acceptance of such tests during software maintenance. Moreover, we capture the developers' perceptions about generated tests. Researchers can use this information for developing new approaches to improve the quality of generated tests, and new test generation tools and tools to remove test smells in existing suites.

1.6 Work organization

The structure of this document is organized as follows. In the Chapter 2, we present the theoretical foundation necessary to understand the content of the work. Chapter 3, is presented the first empirical study on how developers deal with maintenance tasks using automatically generated and manually written tests.

Chapter 4 presents a survey with software professionals, asking their perceptions when comparing original Randoop tests to their refactored versions generated by applying the automatic refactoring techniques presented in Chapter 2. In Chapter 5, we discuss the third study where we replicate the first one, but focus on versions of Randoop tests: original and refactored ones. In Chapter 6, the threats to the validity of this work are presented. In the Chapter 7, we discuss works related to this research . Finally, in Chapter 8, we present conclusions and perspectives for future works.

Chapter 2

Background

In this chapter, we discuss important topics related to our work. In Section 2.1, two automatic unit test generation tools are presented: EvoSuite and Randoop. In Section 2.2, we introduce the concept of Test Smells and some kinds of them. In Section 2.3, the concepts of refactoring and some associated techniques are presented. In section 2.4, we present software maintenance concepts and list their categories of activities. In Section 2.5, the final considerations of the chapter are presented.

2.1 Test Generation Tools

Automatic test generation tools have gained notoriety due to the fact they may be a great alternative to reduce the costs of creating sound test cases. Two of the most well-known test generation tools are Randoop[1] [47] and Evosuite[2] [23].

2.1.1 Randoop

Randoop [47] is a tool that implements feedback-directed random test generation for object-oriented Java programs. Randoop's generation process builds sequences of method calls to exercise the system under testing. It focuses on generating tests that check code elements that could lead to basic contract violations. For instance, a test case that verifies whether a transitive property (*e.g.*, `o1.equals(o2) && o2.equals(o3)` $\rightarrow$ `o1.equals(o3)`)

[1]https://randoop.github.io/randoop/
[2]http://www.evosuite.org/

remains valid after a sequence of method calls. Following construction, it executes the testing sequences to produce results that are used for generating other tests. Figure 3.4 shows an example of Randoop test cases.

Randoop can be used (i) to find bugs, and (ii) to create regression tests to reflect the current behavior of a given program. The Randoop project is still very active. New versions of the tool have addressed complex issues such as the generation of invalid calls to static members, and flaky tests. Several works have used Randoop and its suites in different scenarios (e.g., [64; 41; 42; 48; 63; 65]). In the empirical studies reported in this work, we used Randoop version 4.1.2.

2.1.2 EvoSuite

EvoSuite is a search-based automatic tool for generating JUnit test suites. It generates tests by adding assertions that summarize the current behavior of the system and enables the detection of possible behavior changes. It applies evolutionary algorithms and searches operators such as selection, mutation, and crossover to evolve the test suite. This evolution is guided by a fitness function based on coverage criteria. Figure 1.1 (c) shows an example of an Evosuite test case.

EvoSuite can be used by the command line, or as a plug-in of popular IDEs (*e.g.,* Eclipse). Moreover, EvoSuite has been used on several industrial projects, finding potential bugs (e.g., [24; 22]). Several new and improved versions of this tool have also been released in the past years. In the empirical studies reported in this work, we used EvoSuite version 1.0.6.

2.2 Test smells

Similar to source code, unit tests can also be affected by poor design and programming practices (i.e., smells). The importance to have a well-designed test code was first discussed by Beck [10]. The term test smell was later defined and cataloged by Van Deursen et al. [19]. Test smells resemble code smells. They are anti-patterns from well-established testing practices and guidelines on how test cases should be implemented, organized, and interact with each other. The presence of tests smells may negatively impact the quality of the system [66], hampering the quality and maintenance of a test suite [40], in addition to impairing its performance (e.g., flaky tests [51; 52]). Van Deursen et al. [19] cataloged 11 types of test smells. In addition, other works have defined more than 80 smells [27].

Grano et al. [29] investigated the diffuseness of test smells in automatically generated test suites. They found that Randoop and Evosuite tend to generate a high quantity of two specific test smells (Assertion Roulette and Eager Test). Moreover, Anonymous Test is a test smells type present in all tests generated by Randoop, caused by the use of stub test names (e.g., test1, test2).

In the context of our work, we focus on the three test smells described below:

- **Anonymous Test:** when a test has a meaningless and unclear method name, not expressing the purpose of the test [57] (e.g., Figure 1.1);

- **Assertion Roulette:** when a test has multiple assertions without explanation messages, making it difficult to read, understand, and maintenance the test and to identify the cause of a failing [19] (e.g., Figure 1.1 (b), lines 6 and 7);

- **Eager Test:** when a test checks several methods of the class to be tested, making it difficult to understand the test target [19] (e.g., Figure 1.1 (a), lines 8 and 9).

2.3 Refactoring Test Cases

Refactoring is the controlled process of modifying a program to improve its code structure without changing its external behavior [21]. The refactoring activity is known to remove code smells and improve code quality aspects such as readability, confine source code complexity, decrease coupling, and increase cohesion [43]. Fowler proposes a catalog of different refactoring types [21]. Among the most popular refactorings, we list Rename Method and Extract Method. The first is used when the name of a method does not explain what it does, therefore it should be properly renamed. The Extract Method can be used to reduce complexity and improve the readability of code. For that, one you move a fragment of code from an existing method into a new method with a representative name.

Refactorings can also be used in test code. Van Deursen et al. [19] define test refactorings as transformations of test code that: i) do not add or remove test cases, and (2) make test code more understandable/readable, and/or maintainable. Therefore, refactoring can be used to remove test smells. For instance, by renaming a test case with a more representative name (See Figure 4.3), a tester may better understand its purpose (removing the Anonymous Test smell). By performing a series of Extract Methods based on test assertions one may reduce, or remove, the Assertion Roulette and Eager Test smells, and have less trouble locating/fixing a bug [40] (See Figure 4.4).

2.4 Software Maintenance

IEEE Standard 1219 [1] defines software maintenance as *"The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment"*. ISO/IEC 12207-95 [46] provides a similar definition: *"The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity."*. Thus, when performing maintenance activities, developers aim to improve the quality of a given code.

Software maintenance is an important topic in software development because (i) it consumes a large part of the overall life-cycle costs and (ii) the inability to change software quickly and reliably means that business opportunities are lost [11].

Maintenance activities can be categorized into four classes [37]:

- **Adaptive:** needed activities to adapt the software to environment changes (e.g., changes in the operating system, hardware, software dependencies), or changes related to organizational policies or rules or legislation (e.g., Brazilian General Law for the Protection of Personal Data);

- **Perfective:** activities that aim to evolve or add functionality to the system, based on needs identified by users when interacting with it (e.g., a feature to improve user experience), or remove unused, ineffective, or nonfunctional features that do not contribute to the expected users' goals;

- **Corrective:** activities that aim to fix software bugs, which can affect its various parts (e.g, design, logic, and code). The need for these corrections can be requested/identified by the software users, as well as by the development team itself, which seeks to resolve them before reaching the users;

- **Preventive:** activities related to reducing the risks associated with the long-term operation of the software, making it more stable, understandable, sustainable, and preventing its deterioration. Code optimization and documentation update are examples of these activities.

In our work, we focus on corrective activities, responsible for around 21% of the maintenance effort [11].

2.5 Final Considerations

In this chapter, two of the most well-known test generation tools were presented. Both of them were used in this work. We discussed the test smells concept and presented three test smells types. We presented the software maintenance concept and its importance to software development. Finally, we explained refactoring, and how two refactoring techniques can be used to reduce test smells.

In the next chapter, we will present the first empirical study of this work, which was designed to compare how developers perform maintenance tasks with automatically generated (Evosuite vs Randoop) and manually-written tests.

Chapter 3

A Study On The Use Of Generated Tests To Guide Maintenance Tasks

In this chapter, we present a study to compare how the developers perform maintenance tasks with automatically generated (Evosuite vs Randoop) and manually-written tests. In section 3.1 we present the motivation to perform this experiment. The research questions, study objects, and study procedure are described in section 3.2. In section 3.3, we present and discuss the found results and their implications. Finally, in section 3.4, we discuss the final considerations of this chapter.

3.1 Motivation

Software maintenance involves a series of tasks (e.g., inspecting, modifying, and updating code artifacts) and it is known as complex and costly [11; 28]. To reduce the risks involved in those tasks, developers often use test cases to guide them to identify and correct undesired modifications. However, it is unclear whether generated tests can be as effective and useful as manually written ones in these tasks.

To evaluate how developers deal with maintenance tasks using automatically generated and manually written tests, we ran this study [31].

3.2 Design and Research Questions

Shamshiri et al. [61] performed a study with students to investigate how they deal with maintenance tasks using Evosuite and manually written tests. In our study, we extended Shamshiri et al.'s work [61] by performing a similar investigation. However, instead of students, we ran our study with 20 developers and we deal with a more comprehensive set of testing generation strategies: manual, Evosuite tool, and Randoop tool. Although Shamshiri et al. did not consider Randoop in their investigation, we decided to include it in our study because other works have attested its practical benefits (e.g., [48; 63]) and have been used as a baseline in the SBST Java Unit Testing Tool Contest [7; 34; 59]. Thus, our study may complement Shamshiri et al.'s work by introducing a more practical scenario (real developers) and by comparing manually written testing with two types of generated tests (Randoop and Evosuite).

This study aims at investigating the performance and perception of developers on performing maintenance tasks (codefix or testfix) guided by failing unit test cases created using different strategies (manually written or automatically generated). Our goal is to understand the outcomes and difficulties that developers have when facing a test failure and need to identify the fault and fix it, which can be related to either code or test malfunction. Our goal is to understand the outcomes and difficulties that developers have when they have failing test cases to help to identify and solve a given fault that could be related to either code or test malfunction. Moreover, we would like to understand whether generated test cases, using different strategies, are a good fit in this scenario.

To guide our investigation, we defined the following research questions[1]:

- **RQ1: Do generated tests influence the effectiveness of developers in determining the source of a problem?** This research question aims to understand whether generated tests help developers to determine the source of a problem and which kind of test is the best option;
- **RQ2: Are generated tests effective to help developers find proper fixes?** This research question aims to investigate whether developers produce correct fixes when using a specific kind of test (manually written, Evosuite, Randoop);

[1]We adapted Shamshiri et al.'s research questions [61]

- **RQ3: Does it take longer to execute maintenance tasks when using generated tests instead of manually written ones?** This research question aims to investigate if generated tests, and which kind, reduce the time needed for identifying and performing maintenance tasks;
- **RQ4: What is the developers' perception of using generated tests when performing maintenance tasks?** This research question aims to collect developers' opinions on the use of generated tests when compared to manual ones.

3.2.1 Participants Selection and Demographics

To participate in our study, we recruited active developers that satisfied the following criteria: i) have Computer Science (or related areas) degrees; ii) have previous experience in Java development; iii) have previous experience with unit testing (JUnit); and iv) have the availability of 1h 30min (minimum) to participate of the study.

We selected 20 developers from two companies (small and medium-size), three from one company and 17 from the other. They work on eight different projects. The nature of these projects varies from mobile and web applications to IoT and embedded systems[2]. The companies have no relationship with each other and do not have projects in common. All participants were volunteers in our study and they did not receive any incentive for participation.

The participants perform the following roles: developer (11), software engineering (5), tester (3), and software analyst (1). Prior to the study, they responded to a questionnaire. Figure 3.1 summarizes the participants' background information. Most participants have experience with Java programming for at least three years. Although most find unit testing important for software development, they rarely write or run unit tests.

3.2.2 Study Objects

To run our study, we needed faulty implementations and faulty tests (manually and automatically generated). For comparison purposes, we reused the implementations and

[2]Due to confidentiality reasons, we cannot disclose information about the products developed in these projects

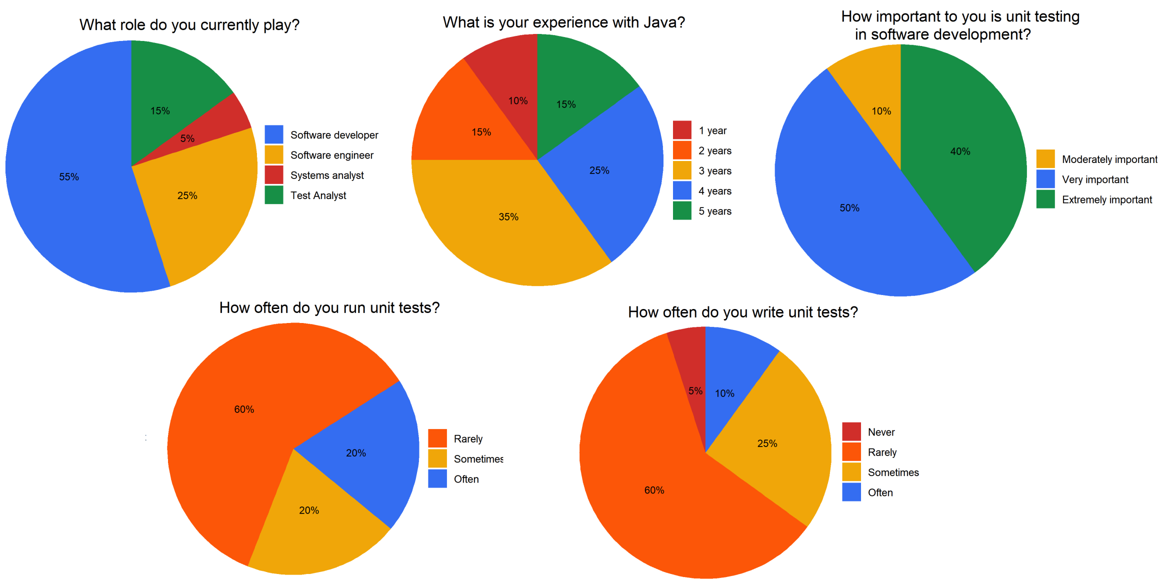


Figure 3.1: Study participantsâ background information.

faults from [61]. In their work, they state these artifacts refer to minimally-faulty implementations, test suites manually-written by real developers, and subtle mistakes. Moreover, all collected faulty versions lead to a single failing test case. Therefore, we worked with two versions (original and faulty) of three classes (`FixedOrderComparator`, `ListPopulation`, and `ComparatorChain`), and their respective failing tests. The `FixedOrderComparator` class is responsible for imposing a specific order on a specific set of objects; `ListPopulation` constructs a genetic population of chromosomes, represented as a List. The `ComparatorChain` class was not used in [61]. However, since we included a new generation tool to the experiment (Randoop), we needed a third class to deal with all treatments. For selecting this class and its respective faults, we applied the guidelines from [58]. `ComparatorChain` runs a series of *comparators* in order to provide a safer comparison for a given pair of objects. Table 3.1 summarizes the characteristics of the used objects.

We injected faults to all three object classes. Those faults were also reused from [61], except the `ComparatorChain` one. All faults emulate subtle real faults reported by other works [61; 63; 64]. To illustrate the injected faults, Figure 3.2 presents the faulty method from ListPopulation (Figure 3.2(a)) and a possible way to fix it (Figure 3.2(b)). This fault refers to the access of a null variable (Figure 3.2(a) - line 4). To fix it, one should properly

Class Name	LOC	Methods	Branch Cov.	Test suite LOC	Manual Test LOC	Evosuite Test LOC	Randoop Test LOC
FixedOrderComparator	98	10	77.5%	137	42	17	26
ListPopulation	97	13	77.3%	149	34	22	28
ComparatorChain	131	18	64.3%	162	29	18	24

Table 3.1: Information about the selected classes. Branch coverage values refer to the test suite used to evaluate the codefixes performed by the participants.

instantiate variable `fitter` (Figure 3.2(b) - line 3).

To complete our study objects we needed Randoop tests for all classes and, Manual and Evosuite test for the `ComparatorChain` class. For that, we proceeded as follows: we ran the Randoop tool for all original versions of the classes, and EvoSuite only for `ComparatorChain`. The tests were generated considering the correct implementations, i.e., prior to any fault injection. Next, we ran the generated suites against the faulty versions and randomly selected a failing one as a representative for the study. Therefore, we selected one Randoop test cases per subject program, and one EvoSuite test for `ComparatorChain` class. For the manual test for the `ComparatorChain` class, we randomly selected a test from the original suite that fails in the faulty version.

As for the faults related to the generated tests, we followed a similar procedure. Figure 3.3 presents an example of a manually written faulty test case (`emptyArray` should be null to trigger the expected exception), and its possible fix. Finally, Figure 3.4 shows a faulty Randoop test (two unknown objects were not properly compared) and its fix.

One may argue that the size of objects is small. However, since we want to compare results to the ones reported by Shamshiri et al., we reused most of their artifacts (classes and faults). They were selected from open-source projects and reflect real-world faults. Shamshiri et al. argue that those artifacts were selected due to their manageable size, availability, and amenability for the research purposes. Those reasons are even more important in our study, because our participants are real developers with limited time. Since we asked the participants to carefully inspect all code (implementation and tests), a more complex configuration (extra classes and test suites with more than a single test) would be impractical.

We also believe the injected faults and scenario (single failing test case) emulate real maintenance tasks. When identifying and fixing faults (source code or test code), developers

```
public Chromosome getFittestChromosome() {
    Chromosome fitter = null;
    for (Chromosome c : this.chromosomes)
        if (c.compareTo(fitter) > 0)
            fitter = c;
        return fitter;
}
```

(a) Faulty version of the `Listpopulation` class. An exception is thrown by `compareTo` since the `fitter` is null.

```
public Chromosome getFittestChromosome() {
-    Chromosome fitter = null;
+    Chromosome fitter = this.chromosomes.get(0);
    for (Chromosome c : this.chromosomes)
        if (c.compareTo(fitter) > 0)
            fitter = c;
        return fitter;
}
```

(b) Possible fix for the fault.

Figure 3.2: Faulty version and possible fix for the `ListPopulation` class.

often focus on a single class and/or small edits. For instance, 47% of the bug-fixes from Defects4J require two or fewer lines of code [33].

3.2.3 Study Procedure

The procedure of our study goes as follows. Prior to the sections, each participant answered a questionnaire about her background. Moreover, the first author ran a brief tutorial about the study and tasks. Each participant was asked to perform three maintenance tasks. Each maintenance task refers to a single fault to be identified and fixed. The fault could be either related to implementation or test code. Moreover, since each participant was asked to perform three maintenance tasks, we vary the type of tests to be used (manually written, Evosuite, and Randoop). Both the task type (*codefix* or *testfix*), the order, and the received test (manually written, Evosuite, or Randoop) were randomly assigned for counterbalancing. To support each task, we provided a pre-configured environment that included an Eclipse IDE and the artifacts to perform the assigned tasks: a project with the class implementation (faulty or not) and a failing test (faulty or not).

```
public void test() {
    try {
        Object[] emptyArray = {};
        FixedOrderComparator comparator = new FixedOrderComparator(emptyArray);
        fail("Exception was supposed to be thrown!");
    } catch (IllegalArgumentException e) {
        assertTrue(true);
    }
}
```

(a) Faulty test that were manually written for `FixedOrderComparator`. `emptyArray` should be null to trigger the exception.

```
public void test() {
    try {
-        Object[] emptyArray = {};
+        Object[] emptyArray = null;
        FixedOrderComparator comparator = new FixedOrderComparator(emptyArray);
        fail("Exception was supposed to be thrown!");
    } catch (IllegalArgumentException e) {
        assertTrue(true);
    }
}
```

(b) Possible fix for the fault.

Figure 3.3: Example of manually written faulty test case for the `FixedOrderComparator` class.

```
1 public void test () throws Throwable {
2     collections.comparators.FixedOrderComparator fixedOrderComparator0 = new collections
      .comparators.FixedOrderComparator ();
3     fixedOrderComparator0.setUnknownObjectBehavior ((int) (byte) 1);
4     int int6 = fixedOrderComparator0.compare ((java.lang.Object) "hi!", (java.lang.
      Object) "");
5     org.junit.Assert.assertTrue ("'" + int6 + "'!='" + 0 + "'", int6 == 1);
6 }
```

(a) Faulty version of a Randoop test. Since two unknown objects are compared in line 5, `int6` should be 0.

```
1 public void test () throws Throwable {
2     collections.comparators.FixedOrderComparator fixedOrderComparator0 = new collections
      .comparators.FixedOrderComparator ();
3     fixedOrderComparator0.setUnknownObjectBehavior ((int) (byte) 1);
4     int int6 = fixedOrderComparator0.compare ((java.lang.Object) "hi!", (java.lang.
      Object) "");
5 -    org.junit.Assert.assertTrue ("'" + int6 + "'!='" + 0 + "'", int6 == 1);
6 +    org.junit.Assert.assertTrue ("'" + int6 + "'!='" + 0 + "'", int6 == 0);
7 }
```

(b) Possible fix for this fault.

Figure 3.4: Example of Randoop generated faulty test case for the `FixedOrderComparator` class.

Our study worked with the following maintenance tasks:

- $< codefix, manual >$: Faulty implementation and a correct manually written failing test;
- $< codefix, evosuite >$: Faulty implementation and a correct Evosuite failing test;
- $< codefix, randoop >$: Faulty implementation and a correct Randoop failing test;
- $< testfix, manual >$: Correct implementation and a faulty manually written failing test;
- $< testfix, evosuite >$: Correct implementation and a faulty Evosuite failing test;
- $< testfix, randoop >$: Correct implementation and a faulty Randoop failing test.

To minimize learning effects, no participant was assigned to the same pair <*fix type, test type*> or class across sessions. For instance, a participant that first was assigned to a $< codefix, manual >$ on `FixedOrderComparator` class, then should be assigned to different tasks and classes in the second and third assignments (e.g., $< codefix, evosuite >$ on `ListPopulation` class, and $< testfix, randoop >$ on `ComparatorChain` class).

The participants were asked to perform each maintenance task with a time limit of 60 minutes and to verbalize and answer a form when identifying the problem. Although in practice one might identify and fix a fault in an intertwined manner, we decide to analyze these tasks separately. Thus, when participants made wrong decisions (e.g., identifying a *testfix* task when in fact should be a *codefix* one), we revealed the correct answer in order to prevent that faults wrongly identified would impact/invalidate the fault fixing task. Only after that, the participants proceed to fix the faults. All sections were performed in person, conducted by the first author, and video-recorded for later analysis. Finally, at the end of the section, the participants were asked to answer a survey questionnaire in which we asked questions related to the maintenance tasks and possible challenges. Figure 3.5 depicts the procedure of our study for a single participant (a more detailed version could be found in Appendix A).

It is important to highlight that we did not impose any protocol for detecting/fixing the faults. When performing the maintenance, a participant could either inspect the implementation code, the test code, or both. Moreover, as either the implementation or test code was

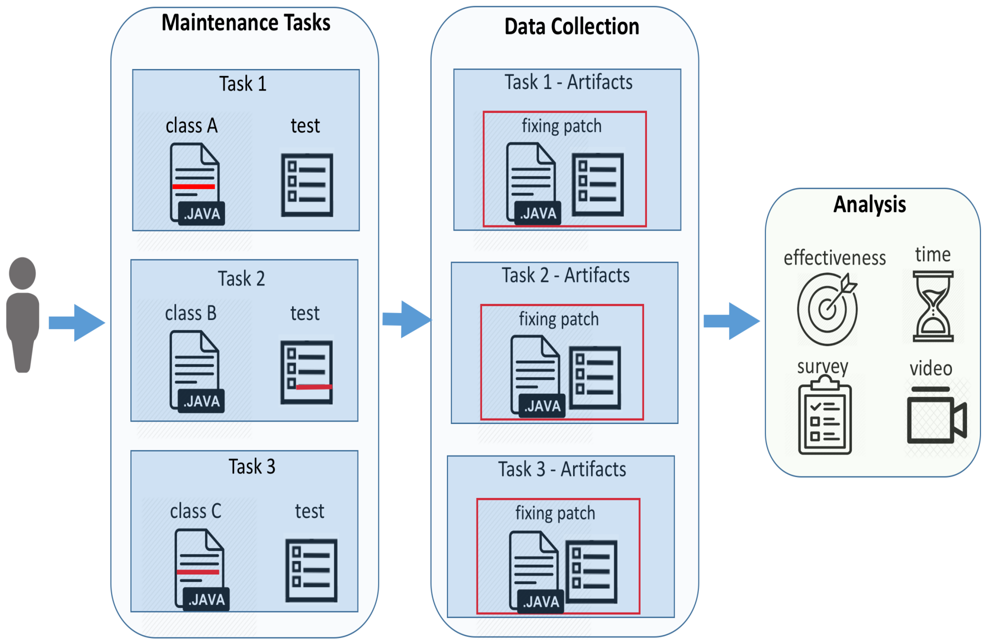


Figure 3.5: Overview of our study procedure for each participant.

faulty, each method included a Javadoc specification. We provided this information to help participants figure the expected behavior of the methods and, therefore, to avoid misleading conclusions based on the code alone.

After the sessions, we evaluated the output artifacts of each participant. To decide weather a codefix or testfix solution was correct, we ran the process in Figure 3.6. A codefix was classified as correct if it did not break any additional test from the original test suite and satisfied a manual inspection. In this manual inspection, we compared the participant's output class with our golden solution and javadoc documentation. For test-fixing tasks, we created reference solutions based on the original test purpose. Then, the first author ran the tests and inspected the code to classify the fix. All artifacts, including the original tests and reference solutions are available on our website[3].

3.3 Results and Discussion

Here, we discuss the collected data and its implications for each research question.

[3]https://github.com/WesleyBrenno/generated-tests-in-the-context-of-maintenance-tasks-a-series-of-empirical-studies/tree/main/study-with-generated-tests

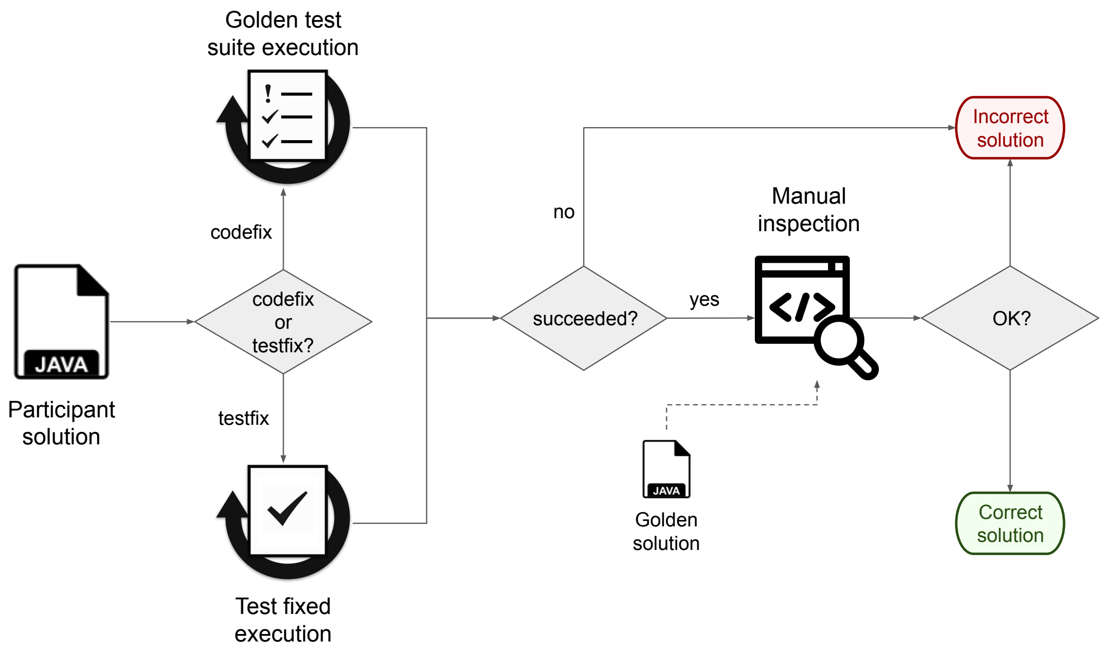


Figure 3.6: Protocol for defining correct code and test fixes.

3.3.1 RQ1: Do generated tests influence the effectiveness of developers on determining the source of a problem?

To answer this question, we observed the participants' effectiveness at identifying whether the faults were in the implementation or test code. Table 3.2 summarizes the results of this investigation. The first three lines refer to results considering all classes, while the remaining present the results per class. As we can see, the tasks helped by Evosuite tests presented very high rates considering *all* (95%), *codefix* (100%), and *testfix* tasks (90%). Those values were greater than the manual (65%, 50%, and 80%) and Randoop ones (50%, 30%, and 70%).

To our surprise, Evosuite tests performed better than the other two strategies in 11 of the 12 analyzed scenarios. This fact may evidence that this type of generated tests could be a good fit for identifying maintenance tasks, even better than manual tests. Developers found those tests easy to follow and helpful when identifying bugs. On the other hand, Randoop tests performed quite poorly. Since Randoop tests focus on contract checking, they tend to be less readable. Therefore, we believe that participants had a hard time understanding the tests and consequently ended up wrongly blaming them (effectiveness for *codefix* was only 30%).

To measure statistical significance when comparing treatments, we first ran the Shapiro-Wilk [56] normality test that did not confirm a normal distribution. Therefore, we used the non-parametric Fisher's exact test [20] for comparisons of correctness. Thus, with a confidence of 95%, we were able to rank the strategies considering the results for *all*, and *codefix* and *testfix* tasks, individually. Table 3.3 summarizes this analyses. $A > B$ indicates that strategy A performed better than B, while $A = B$ says they are statistically equivalent. Then, Evosuite strategy performed better considering *all* tasks together and only *codefixes*, but it was similar to the others when considering *testfixes*. On the other hand, Manual tasks performed similarly to Randoop ones in all three analyses.

Thus, we can answer RQ1 by saying that, in general, developers were more accurate at identifying maintenance tasks when using Evosuite tests, while they were equally accurate when using manually written and Randoop tests. Those results go against Shamshiri et al.'s findings [61], in which they state there is no difference to using manual or generated tests. Our results show that not only there is a difference between those tasks, but also the type of generation tool used seems to play an important role.

***RQ1**: Developers were more accurate at identifying maintenance tasks using Evosuite tests and equally accurate using manually written and Randoop tests.*

3.3.2 RQ2: Are generated tests effective to help to find proper fixes?

To answer this question, we followed the protocol defined in the end of Section 3.2.3 and presented in Figure 3.6). Table 3.4 presents the collected results.

In general, we observed that participants were similarly effective at producing correct fixes using the three strategies (manual, Evosuite, and Randoop). Our statistical analysis reinforces these conclusions (Table 3.5). The only exception was *codefix* for Randoop where only one participant was able to fix the fault using a failing Randoop test. Again, to our surprise, manually-written tests did not perform better than generated ones. Thus, we can answer RQ2 by stating that, in general, generated tests are as effective as manually written ones to help to find proper fixes, regardless of the tool.

***RQ2**: Generated tests are as effective as manually written ones to help to find proper fixes.*

Task type	Class	Manual	Evosuite	Randoop
all	all	13/20 (65%)	19/20 (95%)	10/20 (50%)
codefix	all	5/10 (50%)	10/10 (100%)	3/10 (30%)
testfix	all	8/10 (80%)	9/10 (90%)	7/10 (70%)
all	FixedOrderComparator	1/5 (20%)	7/7 (100%)	6/8 (75%)
codefix	FixedOrderComparator	1/3 (33%)	3/3 (100%)	2/4 (50%)
testfix	FixedOrderComparator	0/2 (0%)	4/4 (100%)	4/4 (100%)
all	ListPopulation	6/7 (85%)	6/6 (100%)	2/7 (28%)
codefix	ListPopulation	2/3 (66%)	4/4 (100%)	1/4 (25%)
testfix	ListPopulation	4/4 (100%)	2/2 (100%)	1/3 (33%)
all	ComparatorChain	6/8 (75%)	6/7 (85%)	2/5 (40%)
codefix	ComparatorChain	2/4 (50%)	3/3 (100%)	0/2 (0%)
testfix	ComparatorChain	4/4 (100%)	3/4 (75%)	2/3 (66%)

Table 3.2: Comparison of correct decisions given Manual, Evosuite or Randoop tests.

Task type	Hiphothesys	p-value	Final ranking
all	Manual = Evosuite	0.0218	Evosuite >Manual = Randoop
	Manual = Randoop	0.5231	
	Evosuite = Randoop	0.0018	
codefix	Manual = Evosuite	0.0162	Evosuite >Manual = Randoop
	Manual = Randoop	0.6499	
	Evosuite = Randoop	0.0015	
testfix	Manual = Evosuite	0.5	Evosuite = Manual = Randoop
	Manual = Randoop	0.291	
	Evosuite = Randoop	1	

Table 3.3: Statistical analysis and ranking considering correct decisions for maintenance tasks.

Task type	**Class**	**Manual**	**Evosuite**	**Randoop**
all	all	14/20 (70%)	14/20 (70%)	9/20 (45%)
codefix	all	6/10 (60%)	8/10 (80%)	1/10 (10%)
testfix	all	8/10 (80%)	6/10 (60%)	8/10 (80%)
all	FixedOrderComparator	2/5 (40%)	5/7 (71%)	5/8 (62%)
codefix	FixedOrderComparator	1/3 (33%)	1/3 (33%)	1/4 (25%)
testfix	FixedOrderComparator	1/2 (50%)	4/4 (100%)	4/4 (100%)
all	ListPopulation	5/7 (71%)	3/6 (50%)	1/7 (14%)
codefix	ListPopulation	3/3 (100%)	2/4 (50%)	0/4 (0%)
testfix	ListPopulation	2/4 (50%)	1/2 (50%)	1/3 (33%)
all	ComparatorChain	7/8 (87%)	6/7 (85%)	3/5 (60%)
codefix	ComparatorChain	4/4 (100%)	3/3 (100%)	0/2 (0%)
testfix	ComparatorChain	3/4 (75%)	3/4 (75%)	3/3 (100%)

Table 3.4: Comparison of correct fixes using Manual, Evosuite or Randoop tests.

Task type	**Hiphothesys**	**p-value**	**Final ranking**
all	Manual = Evosuite	1	Evosuite = Manual = Randoop
	Manual = Randoop	0.2000	
	Evosuite = Randoop	0.2000	
codefix	Manual = Evosuite	0.3142	Evosuite = Manual >Randoop
	Manual = Randoop	0.0572	
	Evosuite = Randoop	0.0027	
testfix	Manual = Evosuite	0.9296	Evosuite = Manual = Randoop
	Manual = Randoop	1	
	Evosuite = Randoop	0.9296	

Table 3.5: Statistical analysis and ranking considering correct fixes for maintenance tasks.

3.3.3 RQ3: Does it take longer to execute maintenance tasks when using generated tests instead of manually written ones?

For this analysis, we evaluated the time participants took to decide whether the maintenance tasks were *codefix* or *testfix*. We also measured the time taken by the participants to perform the fixes.

Figure 3.7 presents the boxplots of the time spent for the participants to decide the tasks. On average, participants took 21 minutes to identify the faults. To better analyze the data, we first ran the Shapiro-Wilk test that indicated a not normal distribution. Therefore, we used the non-parametric Mann-Whitney U test [6] for comparisons of duration values. This test could not find any significant difference among the strategies when considering all classes neither for *codefix* or *testfix*. However, when observing the classes individually, we could find differences. For instance, participants took, in general, longer time to identify *codefixes* in the *FixedOrderComparator* class using manual tests. On the other hand, Evosuite's tests performed worse for *testfixes* in the *ListPopulation* class.

Figure 3.8 presents the distribution of the time that participants took to fix the faults. On average, participants took 8 minutes to fix the faults. Again, the data does not follow a normal distribution and we used the Mann-Whitney U test for comparisons. The test also did not find any significant difference among the strategies in general. Differences were found when observing the classes individually. For instance, participants took, in general, longer time to identify *codefixes* in the *ListPopulation* class using Evosuite tests, but Randoop tests were very effective when correcting *testfixes* in the same class.

Thus, we can answer RQ3 and state that, in general, we cannot say that it takes longer to execute maintenance tasks when using generated tests, regardless of the used tool. However, the class under maintenance may impact the results. Again, this goes against Shamshiri et al.'s findings [61], which say that manually developers are more efficient at maintenance tasks when using manually written tests. Our study did not find pieces of evidence in this sense.

RQ3*: We cannot say that it takes longer to execute maintenance tasks when using automatically generated tests by tools.*

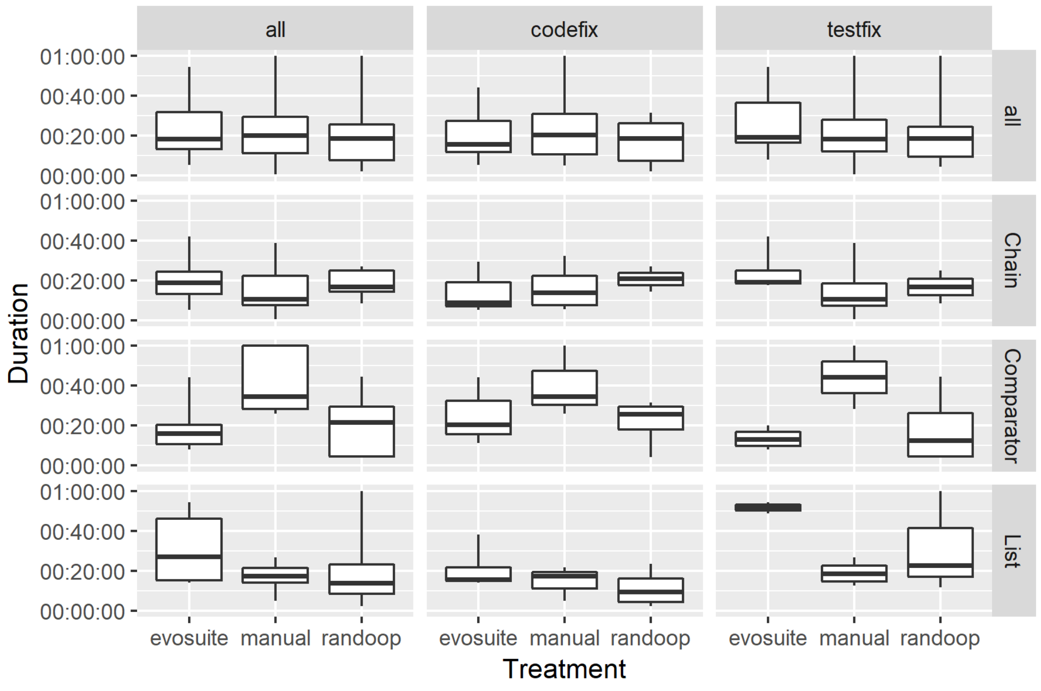


Figure 3.7: The time developers spent to identify their maintenance tasks grouped by task type and class.

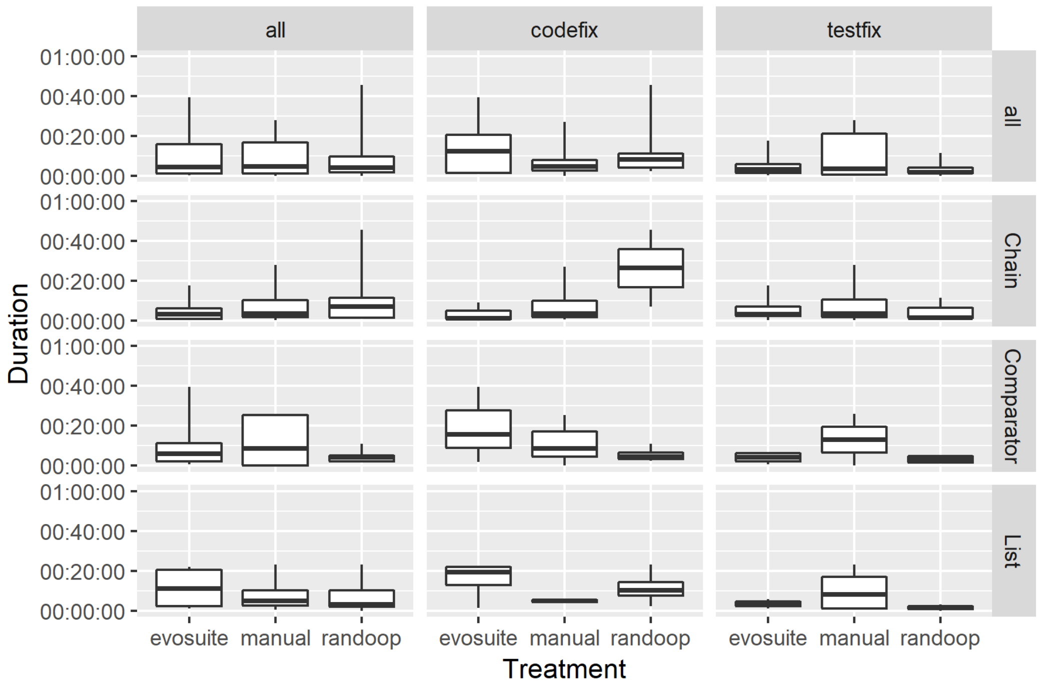


Figure 3.8: The time developers spent to fix their maintenance tasks grouped by task type and class.

3.3.4 RQ4: What is the developers' perception of using generated tests when performing maintenance tasks?

To answer RQ4, we went to the survey responses. Figures 3.9 summarize the answers for *testfix* and *codefix* tasks. In general, participants found the tasks clear (Question 1) and had enough time to finish them (Question 2). Participants found easier to identify the fault type (Question 3) using Evosuite tests for *codefixes*, but not for *testfixes*. On the other hand, Randoop and Manual tests responses were quite similar. These results agree with the analysis and conclusions of Section 3.3.1.

Question 4 asked the participants' perceptions about the activity of fixing the bugs. As we can see, the majority of participants had a similar opinion about manual and Evosuite tasks for *codefixes* and *testfixes*. This goes along with the actual success outcome of the participants (Table 3.4). However, they found it easier to fix bugs using Randoop tests. Although easier to fix, according to participants, the results using Randoop tests were statistically similar (*codefix*) and worst (*testfix*) when compared to the other two strategies.

Participants reported higher confidence in the correctness (Question 5) and quality (Question 8) of their test fixing tasks when using manually written tests, which was expected since generated tests tend to be less readable. However, responses were quite similar for correctness when considering *codefix* tasks and comparing Randoop and Evosuite.

Regarding understanding the class under test (Question 6), for *codefix* tasks, participants found manually written more helpful, followed by Evosuite and Randoop, respectively. For *testfixes*, Manual and Evosuite tests were considered better help than Randoop tests. Possible reasoning for Randoop's poor evaluation is that its test cases focus on basic contract checking, which might not reflect direct documentation of the intended behavior of the program.

Participants found manually written tests easier to understand when used in both *codefixes* and *testfixes*, followed by Evosuite's and Randoop's tests (Question 7). This suggests that participants still find generated tests not ideal to read. Moreover, it reinforces the trend that Randoop tests are hard to inspect due to test smells. For instance, a participant stated the following: *"I had to go back and forth to the code to understand the test's behavior. The test did not have a good name nor its variables, which made it hard to follow"*. In the same sense, a different participant stated: *"The test was full of magic numbers and names that*

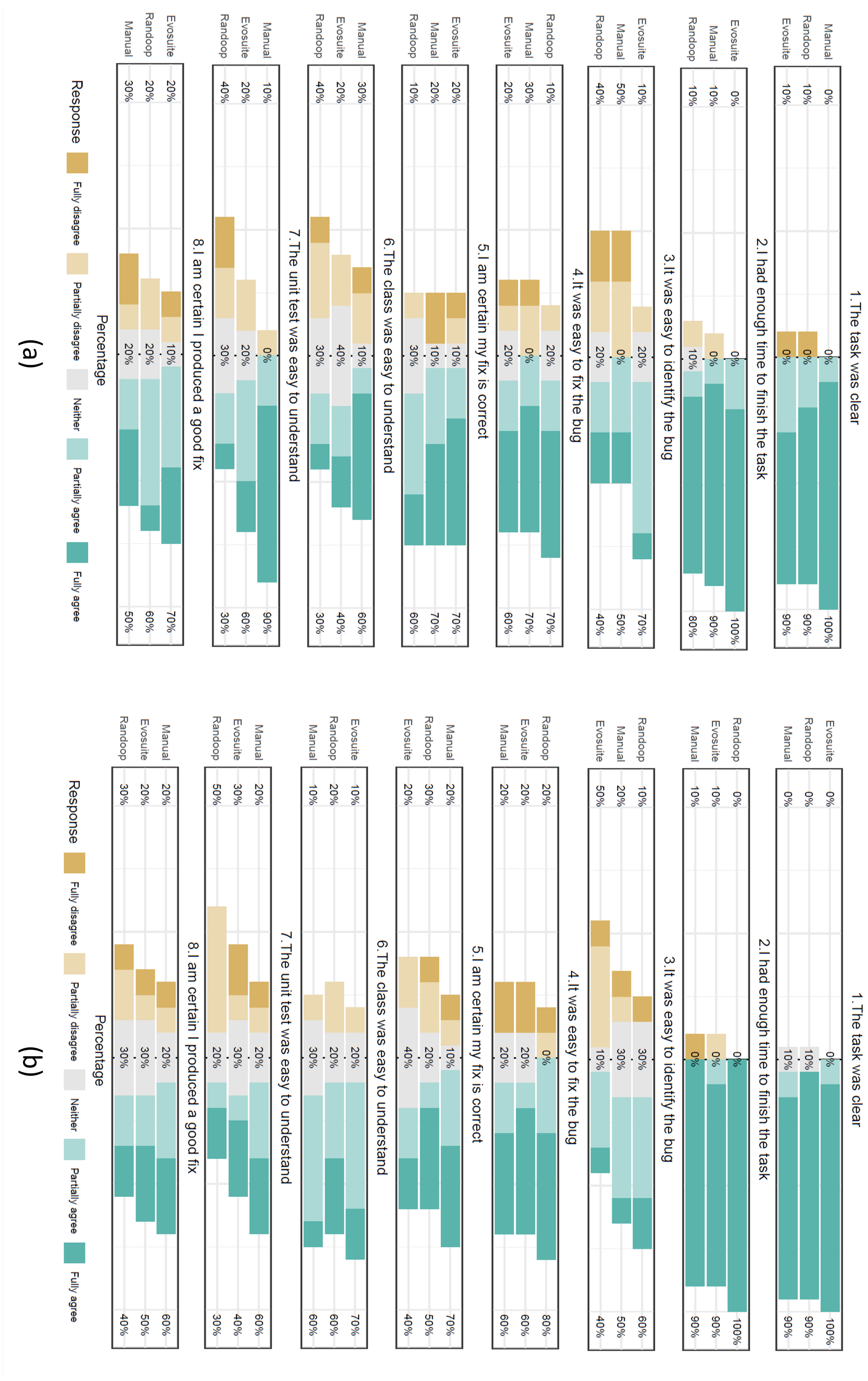


Figure 3.9: Overview of the survey responses relating to (a) *codefix* and (b) *testfix* maintenance tasks.

were not related to the class”.

Finally, we found that participants had lower confidence in the quality of their fixes when using manually written tests to fix a code, but higher confidence when the fault was in the test. Since manually written tests were found easier to understand, it was not a surprise that developers were confident about their *testfixes*. However, they were not so sure about their code fixes. This might reflect that, although easier to read, a manual test often does not help to localize and/or understand code faults. In this sense, generated tests might be a good option. Since they use systematic approaches for test generation, this might guide developers to better understand the code and find its weak spots. Finally, the trend was again confirmed when Evosuite tests were better evaluated than Randoop’s.

RQ4*: Participants found it easier to identify the fault type using Evosuite tests for code fixes, easier to fix bugs using Randoop tests, easier to understand the class under test using manually written for code fixes, and Manual and Evosuite for test fixes, had higher confidence in the correctness and quality of their test fixes tasks when using manually written tests, and found manually written tests easier to understand to both tasks types.*

3.3.5 Analysis by Roles

We reanalyzed the collected data now focusing on the participants’ roles. Regarding RQ1, we found no difference, between the participant’s role in the effectiveness of developers on determining the source of the problem. All of them were more effective when using the Evosuite tests. On the other hand, the *Test analysts* were more effective in fixing the bug using manual tests, while *software developers* and *Software engineers* were more effective using Evosuite tests (RQ2). We also analyzed the results from RQ3 and found that *Test analysts*, *System analysts* and *software engineers* took less time to identify the fault using Randoop tests, while *Software developers* were more efficient using Evosuite tests. *Software developers*, *Software engineers*, and *Test analysts* took less time to fix the bug using the Evosuite tests, while the system analysts were more efficient using the Randoop tests. Nevertheless, due to the small number of developers per role, these results have no statistical relevance.

3.3.6 Diverging Results

The results of this investigation were, in some aspects, surprising and different from Shamshiri et al.'s [61]. For instance, we found differences in the developers' efficiency in identifying bugs. Moreover, developers were less confident about their actions when guided by manual tests. Moreover, Evosuite tests were found as a great help in the maintenance tasks and without imposing more working time. Finally, Randoop tests, although as efficient as the other strategies, did not perform well on the developers' perception, which found them hard to follow.

We see two possible reasons for the diverging results: i) different profiles of participants, and ii) the impact of extra artifacts. Developers are likely to have faced similar tasks in their regular jobs, which might be the reason why they found generated tests useful for bug fixing, though harder to read. Students, on the other hand, are often less experienced, which might be the reason why they performed better with more readable tests (manually written). To assess whether the artifacts introduced in our study (class `ComparatorChain` and Randoop) influenced the results, we ran a side investigation considering a scenario identical to the original study, analyzing the data excluding the related artifacts introduced. The results remain. For instance, we found developers are as effective when performing maintenance tasks using manual or generated tests, Shamshiri et al.'s found they performed better with manual tests. Moreover, they could not find differences in the effectiveness for identifying the type of the bug, we found that developers often perform better when using Evosuite tests. Therefore, we believe the diverging results are mostly due to the different profiles of participants, a more realistic scenario.

3.4 Final Considerations

In this chapter, we presented our first empirical study that compared how real developers perform maintenance tasks with automatically generated (Randoop and Evosuite) and manually-written test cases. Our results found shows that developers were more accurate at identifying maintenance tasks when using Evosuite tests, while they were equally accurate when using manually written and Randoop tests. Moreover, they were similarly effective at producing correct bugfixes using the three strategies (manually written, Evosuite, and Randoop). Regarding their perspectives, developers were more confident that produced correct outputs when using Evosuite tests, but they found manual tests a better proxy for the classes under test's behavior. Those results indicate that automatically generated tests, specially Evosuite's, can be a great help for identifying faults during maintenance. Those results differ from previous findings [61]. Since all strategies were similar at helping to produce correct bug fixes, and with similar efficiency, we can say developers may incorporate generated test suites into their projects at any stage. Moreover, they still find generated tests hard to read, specially Randoop's. The Evosuite test case structure was more appreciated but still needs some improvements.

These results evidence the need for quality improvements in automatically generated test suites, especially Randoop ones. We believe that by solving test smells with well-known refactoring strategies one may improve the quality of them in practice. In the next chapters, we validate this hypothesis with

Chapter 4

A Survey to Evaluate Developers Perspectives on Refactored Tests

In this chapter, we present a survey with developers to access their perception of the use of refactored Randoop tests. In section 4.1, we discuss the motivation to apply this survey. The research questions, study objects, and methodology are described in section 4.2. In section 4.3, we present and discuss the found results and implications. Finally, in section 4.4, we present the final considerations of this chapter.

4.1 Motivation

The study presented in Chapter 3 showed that developers did not evaluate well Randoop tests, mainly due to code quality issues related to test smells. To better understand this issue, we conducted a survey of 82 software practitioners in order to compare their preferences on different versions of Randoop test cases.

4.2 Design and Research Questions

Knowing that refactoring edits can improve code quality, the goal of this survey is to have a deeper understanding of developers' perspectives regarding test names and code, with and without refactoring. To guide this investigation, we define the following research questions (RQ):

- **RQ5**: What is the developers' perception concerning the names of Randoop test cases?
- **RQ6**: Do developers prefer the original Randoop tests or the refactored ones?

4.2.1 Methodology

The survey was presented as a three-section Google Form[1] with three sections:

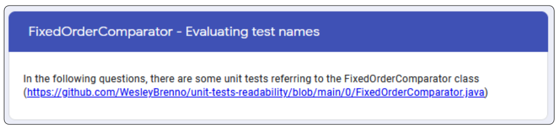
1. **Context and Participant Background**. In this section, we introduced the goal and procedure of the survey and asked questions about the participant's background and her previous experiences with software development and unit testing.

2. **Evaluating Test Naming**. In this section, the participant needed to answer three questions: i) given a test case, indicate the level of agreement with a suggested name. The suggested name was generated using an automatic refactoring strategy (details in Chapter 2); ii) given a test case, select the most appropriate/descriptive name from three options (original Randoop test name; automatic renaming; or a name chosen by an experienced developer); and iii) given a list of candidate names, select the one that fits better for a test that exercises a given code snippet. The candidate list includes the original Randoop test name, an automatically generated one, and a test name chosen by a developer. Figure 4.1 exemplifies questions from this section.

3. **Evaluating Split Tests**. In this section, given four versions of a given Randoop test case (A - original Randoop test; B - split Randoop test; C - renamed Randoop test; and D - split and renamed Randoop test). Versions B-D apply automatic refactorings for reducing the Assertion Roulette and Eager Test smell (details in section 4.2.2). We asked the participant to answer two questions: i) what is the best option?; and ii) which option would you consider including in your test suite? Figure 4.2 exemplifies questions from this section.

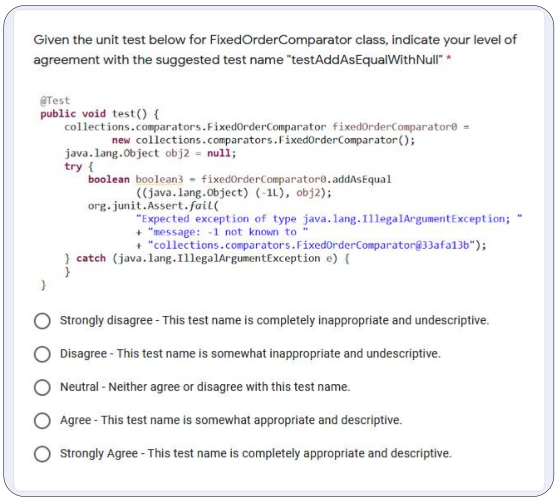
It is important to highlight that, to complement the objective questions, the survey included open questions where participants justified their choices. Moreover, to avoid possible bias related to the used artifacts, we randomly associated participants with classes and test cases. Therefore, we have balanced responses.

[1]https://forms.gle/ESYSZLA5DMQ1PAmx8


FixedOrderComparator - Evaluating test names

In the following questions, there are some unit tests referring to the FixedOrderComparator class (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/0/FixedOrderComparator.java)

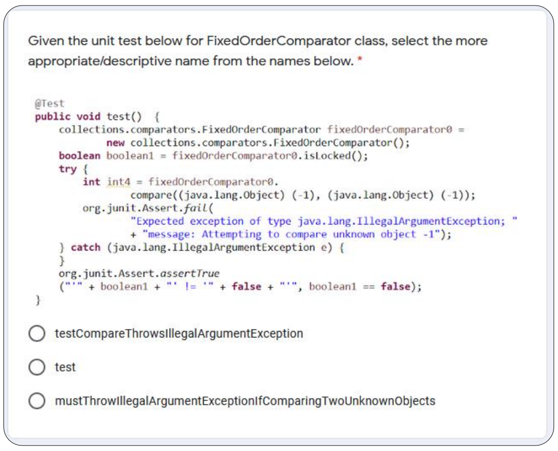
(a) CUT presentation.


Given the unit test below for FixedOrderComparator class, indicate your level of agreement with the suggested test name "testAddAsEqualWithNull" *

```
@Test
public void test() {
    collections.comparators.FixedOrderComparator fixedOrderComparator0 =
            new collections.comparators.FixedOrderComparator();
    java.lang.Object obj2 = null;
    try {
        boolean boolean3 = fixedOrderComparator0.addAsEqual
                ((java.lang.Object) (-1L), obj2);
        org.junit.Assert.fail(
                "Expected exception of type java.lang.IllegalArgumentException; "
                + "message: -1 not known to "
                + "collections.comparators.FixedOrderComparator@33afa13b");
    } catch (java.lang.IllegalArgumentException e) {
    }
}
```

- Strongly disagree - This test name is completely inappropriate and undescriptive.
- Disagree - This test name is somewhat inappropriate and undescriptive.
- Neutral - Neither agree or disagree with this test name.
- Agree - This test name is somewhat appropriate and descriptive.
- Strongly Agree - This test name is completely appropriate and descriptive.

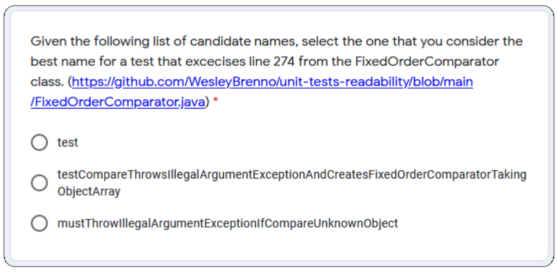
(b) Example of first question.


Given the unit test below for FixedOrderComparator class, select the more appropriate/descriptive name from the names below. *

```
@Test
public void test()  {
    collections.comparators.FixedOrderComparator fixedOrderComparator0 =
            new collections.comparators.FixedOrderComparator();
    boolean boolean1 = fixedOrderComparator0.isLocked();
    try {
        int int4 = fixedOrderComparator0.
                compare((java.lang.Object) (-1), (java.lang.Object) (-1));
        org.junit.Assert.fail(
                "Expected exception of type java.lang.IllegalArgumentException; "
                + "message: Attempting to compare unknown object -1");
    } catch (java.lang.IllegalArgumentException e) {
    }
    org.junit.Assert.assertTrue
    ("'" + boolean1 + "' != '" + false + "'", boolean1 == false);
}
```

- testCompareThrowsIllegalArgumentException
- test
- mustThrowIllegalArgumentExceptionIfComparingTwoUnknownObjects

(c) Example of second question.


Given the following list of candidate names, select the one that you consider the best name for a test that excecises line 274 from the FixedOrderComparator class. (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/FixedOrderComparator.java) *

- test
- testCompareThrowsIllegalArgumentExceptionAndCreatesFixedOrderComparatorTakingObjectArray
- mustThrowIllegalArgumentExceptionIfCompareUnknownObject

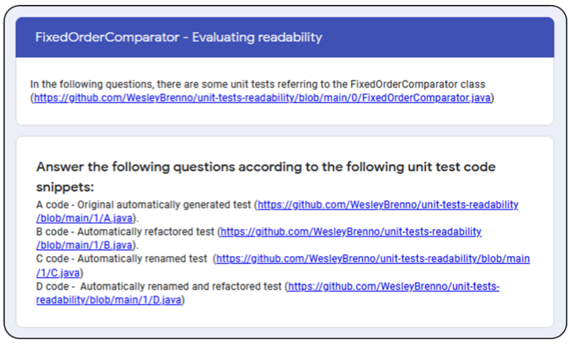
(d) Example of third question.

Figure 4.1: Examples of questions from the second section

4.2.2 Study Objects

In the survey, we presented examples of classes, tests, and test names. For that, we reused the classes (`ListPopulation`, `FixedOrderComparator`, `ComparatorChain`) and Randoop tests from our first experiment (Chapter 3). Moreover, we generated new versions of the tests by refactoring them in order to fix the found test smells (Assertion Roulette, Eager Test, and/or Anonymous Test). The refactorings were applied using the following automatic strategies:

Test Renaming. This is an adaptation of the Rename Method refactoring designed to fix the Anonymous Test smell. Randoop tests receive standard test names (e.g, test23). Ermira et al. [18] propose an approach for generating names specifically for automatically generated tests. It uses coverage goals (method coverage, exception coverage, output coverage, and input coverage) for generating informative names. Coverage goals are a set of distinct objectives, such that a set of tests is considered adequate if, for each objective, there is at least one



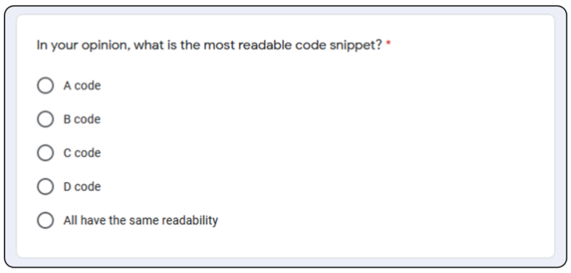
FixedOrderComparator - Evaluating readability

In the following questions, there are some unit tests referring to the FixedOrderComparator class (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/0/FixedOrderComparator.java)

Answer the following questions according to the following unit test code snippets:

A code - Original automatically generated test (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/1/A.java).
B code - Automatically refactored test (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/1/B.java).
C code - Automatically renamed test (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/1/C.java)
D code - Automatically renamed and refactored test (https://github.com/WesleyBrenno/unit-tests-readability/blob/main/1/D.java)

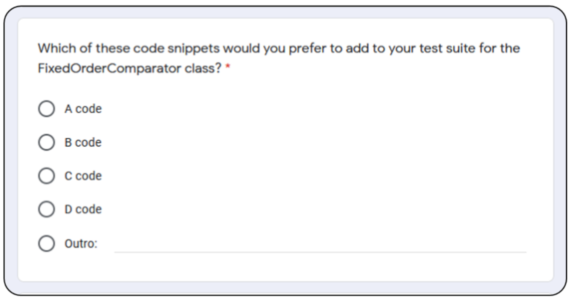
(a) CUT and tests snippets presentation.



In your opinion, what is the most readable code snippet? *

- ○ A code
- ○ B code
- ○ C code
- ○ D code
- ○ All have the same readability

(b) Example of first question.



Which of these code snippets would you prefer to add to your test suite for the FixedOrderComparator class? *

- ○ A code
- ○ B code
- ○ C code
- ○ D code
- ○ Outro:

(c) Example of second question.

Figure 4.2: Examples of questions from the third section.

test that exercises it. While coverage goals may not describe a real test intent, they serve as reasonable approximations as they can describe what a test does. The approach was designed for EvoSuite test cases . Although EvoSuite version 1.0.6, which was used in our first study, include this naming strategy, we did not use it because we are reusing the tests from Shamshiri et. al.'s work [61]. As far as we know, we are the first to adapt and apply this approach for Randoop tests.

As EvoSuite captures coverage during its test-generating process, we extracted its test-generating approach and adapted it to make it work with Randoop tests. Figure 4.3 presents an example of a Randoop test before and after its automatic renaming. To generate the names for each test, the coverage goals for each test are ranked according to the following hierarchy: goals covered in the suite uniquely by this test, exception coverage, method coverage, output coverage, and input coverage. Then, the N best-ranked goals are selected, and the number of selected goals can be configured and is responsible for controlling the size of the generated names, in all our work we use the default value, which is two. Next, the goals selected are converted into text to be concatenated and composed of the test name. Finally, the test names pass by basic processing with a technique based on abstract text summary algorithms,

```
@Test
- public void test23() throws Throwable {
+ public void testSetElitismRateThrowsOutOfRangeExceptionAndToString() throws Throwable
    {
        math.genetics.ElitisticListPopulation elitisticListPopulation2 = new math.
    genetics.ElitisticListPopulation(100, 0.0d);
        java.lang.String str3 = elitisticListPopulation2.toString();
        try {
            elitisticListPopulation2.setElitismRate((double) (byte) -1);
            org.junit.Assert.fail("Expected exception of type math.exception.
    OutOfRangeException; message: elitism rate (-1)");
        } catch (math.exception.OutOfRangeException e) {
        }
        org.junit.Assert.assertTrue("'" + str3 + "' = '" + "[]" + "'",
    str3.equals("[]"));
```

Figure 4.3: Renamed Randoop tests for `ListPopulation` class.

to simplify common patterns to more natural versions.

Splitting Tests. Randoop tests are often long and include a series of asserts (see Figures 4.4). To avoid Assertion Roulette and Eager Test smells, a test can be divided based on their asserts. For that, we implement a script that reuses Eclipse Refactoring API[2]. It receives a given test case t with n assertions and performs a series of Extract Method refactorings, generating n new test cases. Each Extract Method is triggered by a test assertion. For that, we analyze the test case Abstract Syntax Tree (AST) and extract to n new methods each assertion statement along with its dependencies. By using the Eclipse Refactoring Engine we guarantee that the new set of test cases are free of compilation errors and preserves the original behavior. For instance, Figure 4.4 presents the original Randoop test case and its respective split suite.

To automatically apply these refactorings, we create a tool. This tool is an Eclipse plug-in that receives as input a Randoop test suite and automatically finds refactoring opportunities and applies them to improve tests' quality. This tool is available on our website [3]

[2]https://www.eclipse.org/jdt/

[3]https://github.com/WesleyBrenno/generated-tests-in-the-context-of-maintenance-tasks-a-series-of-empirical-studies/tree/main/plugin

```
@Test
    public void test() throws Throwable {
        collections.comparators.FixedOrderComparator fixedOrderComparator0 = new
    collections.comparators.FixedOrderComparator();
        boolean boolean1 = fixedOrderComparator0.isLocked();
        fixedOrderComparator0.setUnknownObjectBehavior((int) (byte) 1);
        fixedOrderComparator0.compare((java.lang.Object) "hi!", (java.lang.Object) "");
        try {
            fixedOrderComparator0.setUnknownObjectBehavior((int) (short) -1);
            org.junit.Assert.fail(
                "Expected exception of type java.lang.UnsupportedOperationException;
            message:  Cannot modify a FixedOrderComparator after a comparison");
        } catch (java.lang.UnsupportedOperationException e) {
        }

    org.junit.Assert.assertTrue("'" + boolean1 + "' != '" + false + "'", boolean1 == false);
    }
```

(a) Original Randoop test example for `FixedOrderComparator` class.

```
@Test()
public void test_1() {
  collections.comparators.FixedOrderComparator fixedOrderComparator0 = new collections.
    comparators.FixedOrderComparator();
  boolean boolean1 = fixedOrderComparator0.isLocked();

    org.junit.Assert.assertTrue("'" + boolean1 + "' != '" + false + "'", boolean1 == false);
}

@Test()
public void test_2() {
  collections.comparators.FixedOrderComparator fixedOrderComparator0 = new collections.
    comparators.FixedOrderComparator();
  boolean boolean1 = fixedOrderComparator0.isLocked();
  fixedOrderComparator0.setUnknownObjectBehavior((int) (byte) 1);
  fixedOrderComparator0.compare((java.lang.Object) "hi!", (java.lang.Object) "");
  try {
    fixedOrderComparator0.setUnknownObjectBehavior((int) (short) -1);
    org.junit.Assert.fail(

    "Expected exception of type java.lang.UnsupportedOperationException; message:  Cannot modify a
    FixedOrderComparator after a comparison");
  } catch (java.lang.UnsupportedOperationException e) {
  }
}
```

(b) Split Randoop tests from (a).

Figure 4.4: Original and split Randoop tests for `FixedOrderComparator` class.

4.3 Results and Discussion

In this section, we present the data analysis and results of the survey.

4.3.1 Participants Demographics

For this study, we recruited volunteers by convenience (contacting developers from partner companies and universities), social networking platforms (e.g., LinkedIn), and snowball sampling [71], i.e., participants were asked to resend the survey invitation to others. The developers have previous experience in Java/JUnit.

We received a total of 82 responses: six graduate students; three master students; two professors; and 71 active developers from various software companies: 51 software engineers, seven software QA analysts, four software analysts, four team leaders, three data scientists, and two requirements analyst. Though this survey was sent to different mailing lists, all participants are from Brazil.

The first section of the survey helped us to understand the participants' backgrounds (Figure 4.5). Most participants have at least three years of experience with Java and create unit tests regularly. Only 20% used test generating tools, mostly Randoop. Next, we answer and discuss RQ5 and RQ6.

4.3.2 RQ5: What is the developers' perception concerning the names of Randoop test cases?

To answer this question, we analyzed the responses of the second section.. In the first question, participants were asked to set their agreement level to the suggested automatically generated name. Figure 4.6 summarized the answers. Most participants (51%) did not find the suggested name suitable for the presented Randoop tests, while 35% agreed with the suggested name. However, by using bootstrap [32], with 95% of confidence, we found no statistical difference to conclude that participants disagree or agree with the proposed names.

Some participants found the suggested names helpful. Here, we list some quotes collected from the open-ended questions: *"The name clearly represents the idea of the test"*; *"The test name already lets me know what will be tested"*; and *"The suggested name pro-*

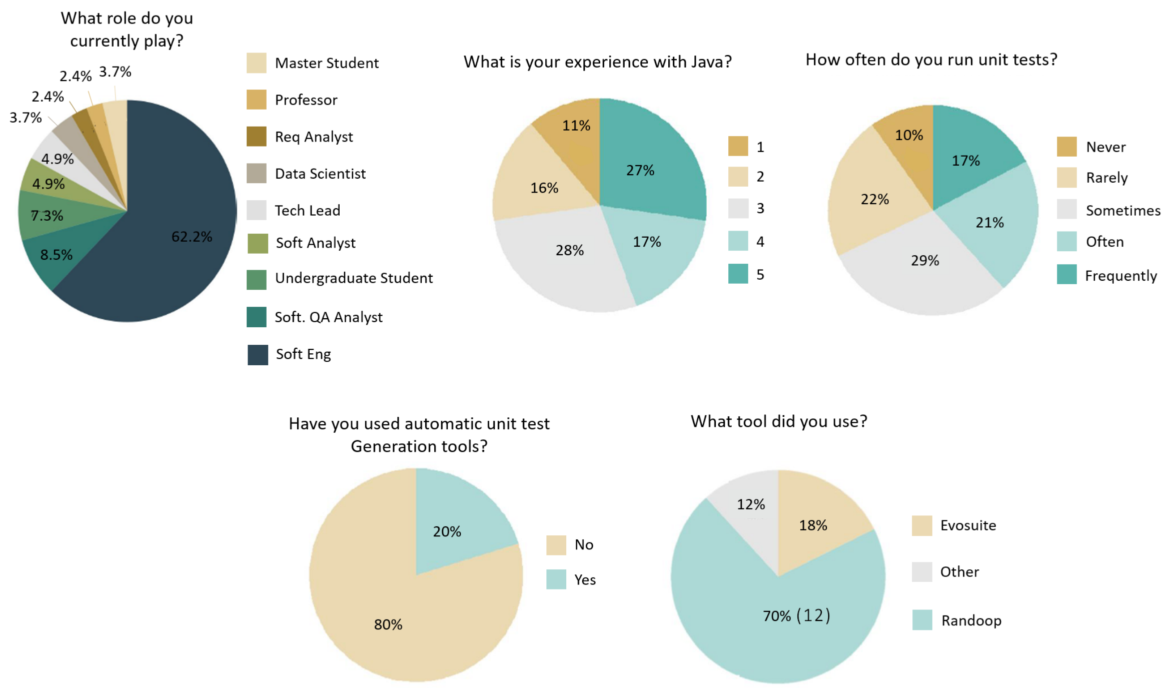


Figure 4.5: Participants' background information.

vides a great improvement when compared to other options".

Although previous works have found significant improvements when renaming Evosuite tests with Ermira et al's technique [18], our results show that although promising, the renaming strategy needs improvement when dealing with Randoop tests. Some participants found the suggested names not descriptive enough for the nature of the test code: *"The test does a lot of internal things which make the name too long. The test should be split into different test cases, so the test names would be more descriptive"*. This quote highlights issues related to Eager test smell and the need for smaller and more focused Randoop test cases. Moreover, some participants disagreed with part of the suggested name: *"One should avoid reserved names such as List, String, Null as much as possible"*.

The second question of this section asked participants to choose the most suitable name for a given Randoop test among three options: a name suggested by an invited developer, an automatic generated name (using the Test Renaming refactoring strategy), and the original Randoop name (*test*). As expected, no participant chose the *"test"* option (Figure 4.7. In general, the participants preferred the names chosen by the invited developers (54%) over the automatically generated names (46%), except for the `ComparatorChain` class (47%

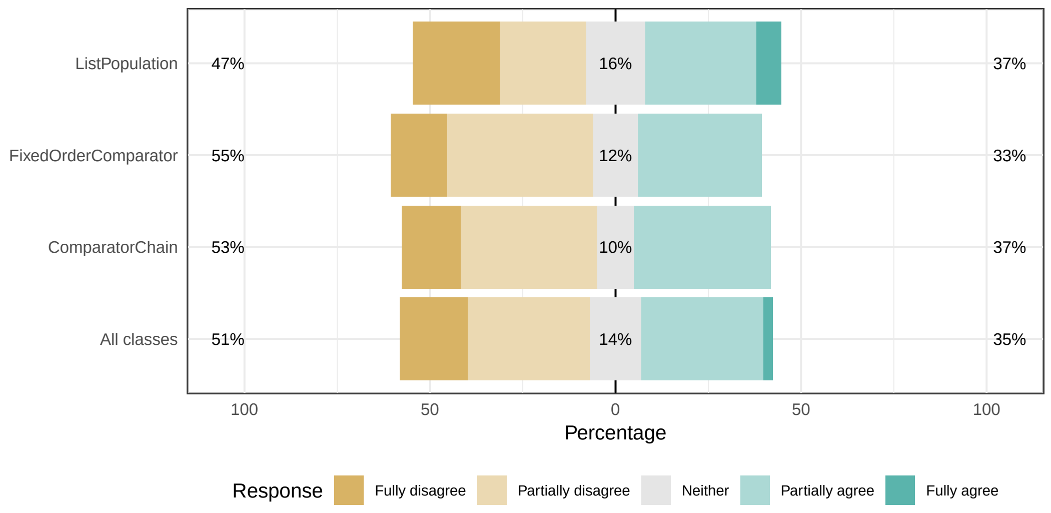


Figure 4.6: Overview of answers about the agreement with automatically generated test names.

and 53%). By using bootstrap, with 95% of confidence, we can not conclude that developers prefer manually written or automatically generated test names. However, participants' comments provide us some reasoning and directions for improving the automatically generated names. A developer that preferred the generated option stated the chosen name was *"very descriptive and reflected the purpose of the test"*. Even when the name was long, one reported: *"the name, despite being long, makes it clear what is being tested and expected results"*. As for the ones who chose the developer's naming, they found the names simpler and easier to understand: *"By reading the names of the test I can easily understand it"*, *"The name reflects the test result and what caused it"*.

In the last question of the second section, we asked participants to select from three options (a name suggested by an invited developer, an automatically generated one, and the original Randoop test name) the best name for a test that would exercise a given code snippet.. Figure 4.8 summarizes the answers. In general, most participants associated the given piece of code under test with the manually written names (61%), followed by the automatic generate names (39%), except for the `ComparatorChain` class (47% and 53%). No participant chose the *"test"* option. Again, our statistical analysis found no difference between the manually chosen and the automatically generated names.

We can then answer RQ5 stating that although there is a numerical advantage for man-

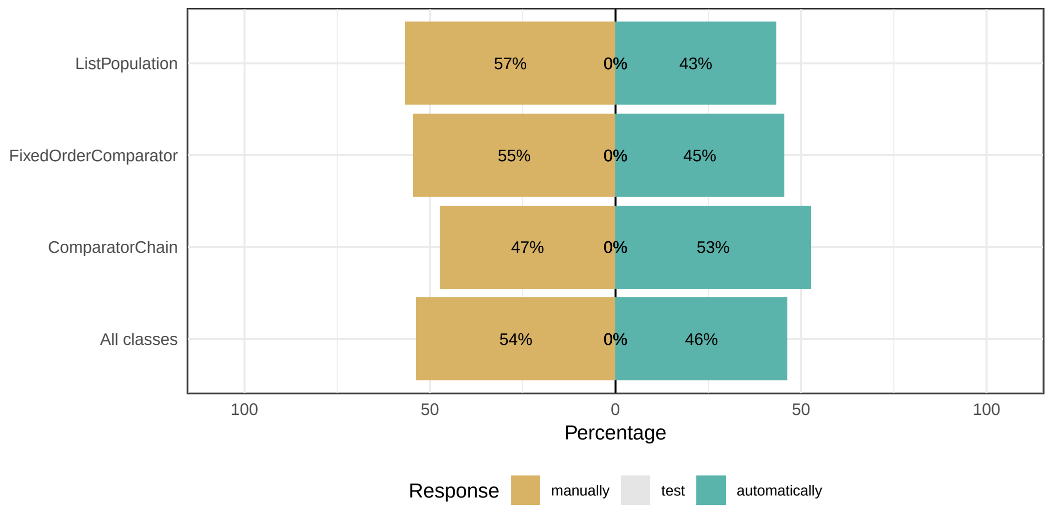


Figure 4.7: Overview of answers about more appropriate test names.

ually chosen test names in our survey, we cannot say that developers prefer them over automatically generated ones. This may indicate that an automatic strategy can be a valuable option for renaming Randoop tests. However, the used renaming strategy is limited and requires further improvements for Randoop tests.

***RQ5**: We cannot say that developers prefer manually chosen test names over automatically generated ones, which indicates that an automatic strategy can be a valuable option for renaming Randoop tests.*

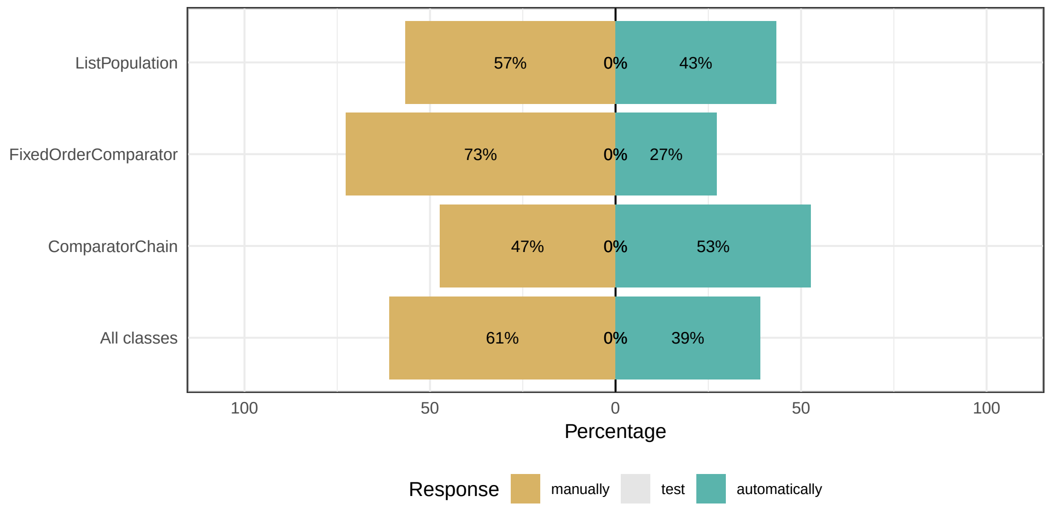


Figure 4.8: Answers about the test that should exercise a CUT.

4.3.3 RQ6: Do developers prefer the original Randoop tests or the refactored ones?

To answer this RQ, we analyzed the responses of the third section. In this section, given a CUT and four test code snippets for it (A - original Randoop test; B - split Randoop test; C - renamed Randoop test; and D - split and renamed Randoop test), we asked participants: i) what is the most readable code snippet?; and ii) which of these codes snippets would you prefer to add to your test suite for the CUT?

Figures 4.9 and 4.10 summarize the collected answers to each question, respectively. Our results showed that most participants find split-renamed tests the best option (84%). Only 2% preferred the option with split tests, 9% the renamed option, 5% found all options similar, and 0% the original Randoop test. Moreover, most participants also prefer to reuse the split-renamed tests (78%). In both cases, with a 95% confidence, there is a significant difference in the developers' preference for Code D (split-renamed tests) to all other options.

The participants' comments on those questions may help us better understand their perspectives. Participants point out that the transformation applied in Code D improves readability and helps fault localization: *"By breaking a test into different methods and renaming them according to their purposes, it makes it easier to read the code. Although the code is more extensive, the simplified parts make help understanding the whole thing"*, *"Although there is code duplicity, when a test fails, one can better understand the reasoning for the fail, because the names are descriptive and the code modularized"*. When choosing option C (Randoop test automatically renamed), participants clearly favor less code duplication: *"more readable code and without redundancies"*. On the other hand, some participants pointed out the refactored versions still need improvement regarding readability: *"Internal variables need to be more descriptive"*, and *"don't use full class names instead of simple names (e.g., don't use collections.comparators.Fixed OrderComparator instead only FixedOrderComparator)"*. Finally, one participant commented: *"Although the names help to identify the purpose of the test, the body of the test is not easy to read. Maintaining these tests might be too costly, therefore, I would avoid their reuse"*.

We can now answer RQ6 by stating that developers prefer the refactored version (split-renamed) that fixes most test smells. However, those versions still require improvements to

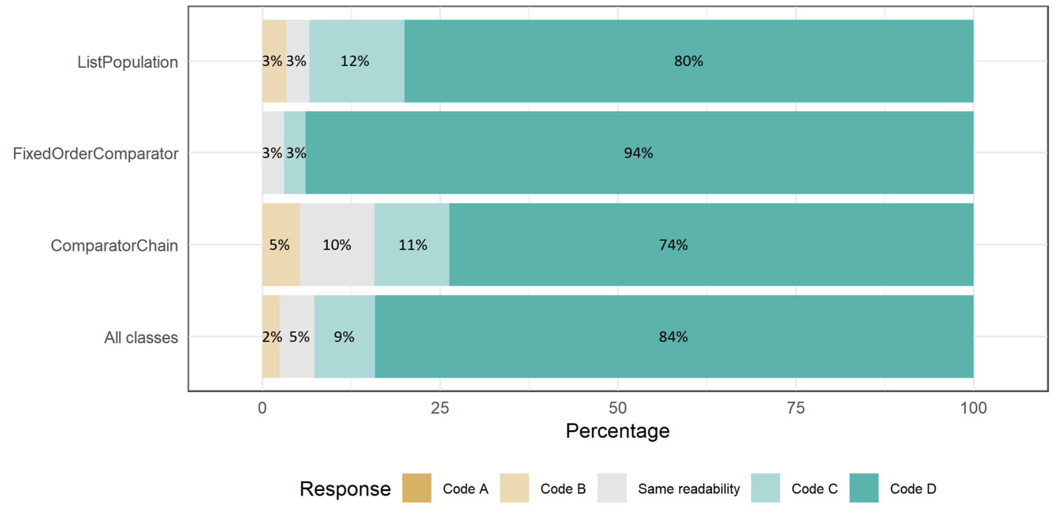


Figure 4.9: Answers about the most readable test code.

be fully accepted. The most cited limitation refers to readability and reuse issues, such as the need for variables renaming and avoiding code duplicities.

***RQ6**: Developers prefer refactored (split-renamed) Randoop tests over the other alternatives (original, only renamed, and only split tests).*

4.3.4 Analysis by Roles

When we analyzed the results by role, we found no significant differences regarding the code readability questions. Refactored codes are considered more readable and eligible to compose a test suite in all scenarios. On the other hand, there was a difference between the participant's answers of each role regarding their preference about manually chosen test names and automatically generated ones for the Randoop tests. *Software engineers*, *technical leaders*, *undergraduate* and *systems analysts* did not find the suggested name suitable for the presented Randoop tests. While *master students* and *quality analysts* agreed with the suggested names. *Software engineers* may be more careful with test names because they work directly with activities influenced by names, such as software evolution and maintenance tasks. *Data scientists*, *professors* and *requirement analysts* did not present an opinion about this topic. Notice that these results may be influenced by the sample sizes instead of the participant roles. Again, due to the small number of developers from each role, these results have no statistical relevance.

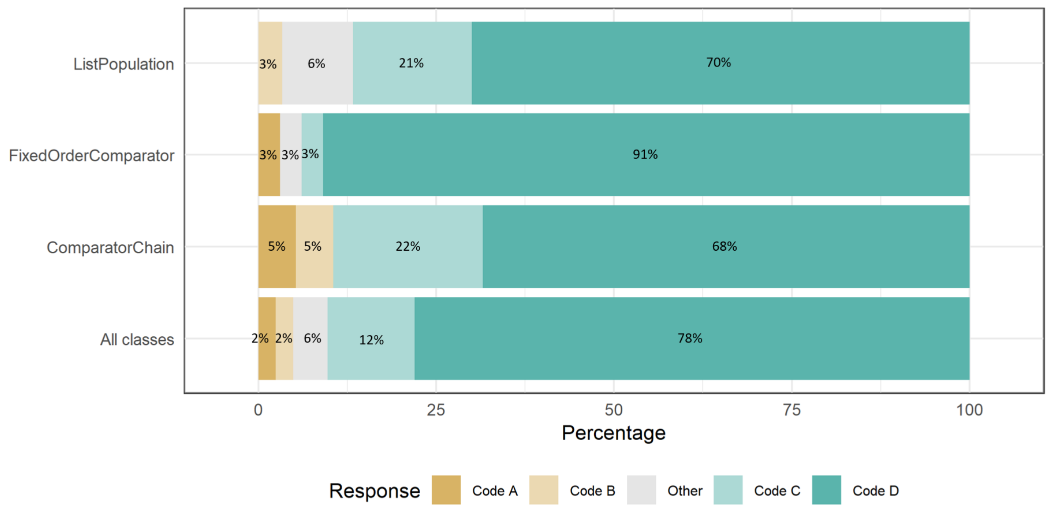


Figure 4.10: Answers about the most appropriate test to include into a test suite.

In practice, often software engineers and quality analysts actually deal with unit tests. Thus, ran a side investigation by filtering our results considering only these two roles. However, we found no significant difference regarding to the code readability questions, as well as for test names questions. The graphs for this analysis can be found in Appendix B.

4.4 Final Considerations

This Chapter presented a survey with 82 developers that assessed developers' perception of the use of refactored Randoop tests. The results shows that: (1) Although we cannot say that developers preferred automatically generated test names over manually written ones, the automatic renaming of Randoop tests was well-received; (2) Developers preferred refactored Randoop tests over original ones. However, in order to fully accepted them, they indicated the need for extra refactorings, such as variables renaming, and extract method. These results motivated us to access the practical impact of the refactorings on the performance of maintenance activities. For this, we performed a third empirical study, where we replicated the first one (Chapter 3), now focusing on different versions of Randoop tests (original and refactored). This study is presented in the next chapter.

Chapter 5

An Study on the Use of Refactored Generated Tests to Guide Maintenance Tasks

In this chapter, we rerun our first study (Chapter 3.3.1), now focusing on different versions of Randoop tests. In the section 5.1, we present the motivation to this replication. The research questions, study objects, study procedure, and follow-up interviews, are described in section 5.2. In the section 5.3, we present and discuss the found results and their implications. Finally, in the section 5.4, we present the final considerations of this chapter.

5.1 Motivation

The survey results (Section 4.3) indicate that Ermira et al.'s [18] renaming strategy can be a plausible solution for replacing stub names, but it is not enough to improve the test quality. On the other hand, developers' perception was that split-renamed tests were the best option and greatly improve test code readability. However, we still have to access how those transformations impact performance in maintenance activities. For this, we ran a new empirical study.

In the study reported in Chapter 3, we investigated the performance and perception of developers when performing maintenance activities when guided by manual and generated test suites (Evosuite or Randoop). Here, we replicate this study focusing on different versions of Randoop tests. The goal of the new study is to investigate the effectiveness of refactored Randoop tests when used for maintenance.

5.2 Design and Research Questions

In this study, developers needed to perform maintenance tasks guided by three types of failing Randoop tests: an original Randoop test, a renamed version, and a split-renamed version. The last two were created by applying the refactoring strategies presented in Section 4.2.2 for reducing test smells. Our goal is to understand whether the refactored tests can impact the use of Randoop tests in maintenance activities when they need to identify a fault and fix it.

To guide this investigation we adapted the research questions from our first study:

- **RQ7: Do refactored Randoop tests improve developers' effectiveness in determining the source of an issue?** Here, we want to understand whether refactored generated tests, can better help developers to determine the source of a problem;
- **RQ8: Do refactored Randoop tests improve developers' effectiveness in performing proper fixes?** We aim to investigate whether developers produce correct fixes when using refactored Randoop tests;
- **RQ9: Do refactored Randoop tests improve the developers' performance to execute maintenance tasks?** Here, we investigate if refactored Randoop tests may speed up the process of identifying and performing maintenance tasks;
- **RQ10: What is the developers' perception about using refactored Randoop tests in maintenance tasks?** Here, we collect developers' opinions on the use of refactored generated tests.

5.2.1 Participants Selection and Demographics

For this study, we applied the same selection criteria as the first one (Section 3.2.1). We recruited 24 volunteer active Brazilian developers (20 males and four females) from 10 different companies (12 from company 1, three from company 2, two from Company 3, and one for companies 4-10). It is a different set of participants, i.e., none of them participated in our first study. They came from different companies (small and medium-size) and perform the following roles: software engineering (17), team leader (3), data scientist (2), and test analyst (2). Similar to the first group, these developers work on projects from different natures (e.g., Mobile and Web Applications, Embedded Systems)

Prior to the study, they answered a background questionnaire (Figure 5.1). Half of the participants have had experience with Java programming for at least three years. Although most find unit testing important for software development, they do not write or run unit tests very frequently. Finally, one participant has used test generation tools before.

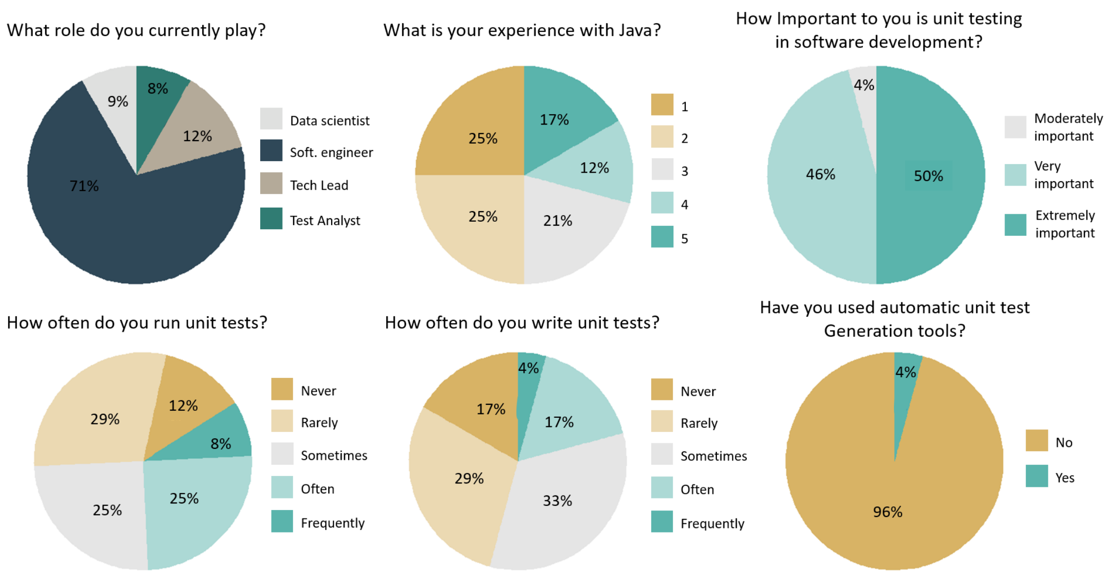


Figure 5.1: Study participants' background information.

5.2.2 Study Objects

To run our study, we reused the implementations and faults from our first study (`ListPopulation`, `FixedOrderComparator` and `ComparatorChain` classes, and Randoop tests), replacing the manual and Evosuite tests by renamed Randoop tests, and split-renamed Randoop tests. To generate refactored Randoop tests we ran the strategies presented in Section 4.2.2. While the original Randoop test and its renamed version refer to a single test, the split-renamed refers to a suite with n tests, where n is the number of asserts from the original Randoop test. Thus, in our study, we rely on the incorrect and faulty implementations of the `ListPopulation`, `FixedOrderComparator` and `ComparatorChain` classes, as well as the correct and faulty Randoop, renamed, and split-renamed tests, for each class.

5.2.3 Study Procedure

This study procedure replicates the one followed in Section 3.2.3 with a single difference: the maintenance tasks. By replacing the manually and Evosuite generated tests with split and split-renamed Randoop suites, our study worked with the following maintenance tasks:

- $< codefix, randoop >$: Faulty implementation and a correct Randoop failing test;
- $< codefix, split >$: Faulty implementation and a correct split Randoop failing test;
- $< codefix, split - renamed >$: Faulty implementation and a split and renamed Randoop failing test;
- $< testfix, randoop >$: Correct implementation and a faulty Randoop failing test;
- $< testfix, split >$: Correct implementation and a faulty split Randoop failing test;
- $< testfix, split{-}renamed >$: Correct implementation and a faulty split and renamed Randoop failing test.

The participants of our study are real developers with limited time. Therefore, to reduce the time required, we opted to not include a treatment related to only renamed tests. It is important to highlight that, this study was run between 2020-2021. Due to imposed COVID restrictions, we adapted our study procedure to work in a totally remote environment. For that, we used tools such as TeamViewer[1] and AnyDesk[2] to provide a controlled environment so the participants could perform the required tasks and Microsoft Teams[3] e Google Meet[4] for remote calls. This configuration allowed the participants to work more flexible hours (the

[1]https://www.teamviewer.com/
[2]https://anydesk.com/
[3]https://www.microsoft.com/microsoft-teams/
[4]https://meet.google.com/

first authors still live-watched all sessions), which enabled us to recruit a relatively larger number of developers this time (24).

All artifacts used in our study, including implementations, faults, tests, and questionnaires, are available on our website [5].

5.2.4 Follow-up Interviews

To complement our analysis, after running the study, we selected a subset of participants and interviewed them with the goal of having a better understanding of the found results. We divided the participants into three groups, according to the type of tests they performed better in the study. For each group, we invited three participants, however, only five were available for the interview (three software engineer, one data scientist and one test analyst). During the interviews, the following questions were asked:

- What difficulties did you experience when deciding the maintenance task to be performed?
- Do you consider the provided test case name good and representative? Did it help you identify/fix the fault?
- Related to tasks with split tests: Did you consider the non-failing tests when identifying/fixing the fault?
- Which of the three treatments (Randoop, renamed, split-renamed tests) would you consider the best option to assist the maintenance activities and why?

All interviews were conducted remotely using either Google Meet[6] or Microsoft Teams[7].

5.3 Results and Discussion

In this section, we discuss the data analysis and results for each of our research questions, and the responses collected during the follow-up interviews.

[5]https://github.com/WesleyBrenno/generated-tests-in-the-context-of-maintenance-tasks-a-series-of-empirical-studies/tree/main/study-with-refactored-generated-tests

[6]https://meet.google.com

[7]https://www.microsoft.com/microsoft-teams/

5.3.1 RQ7: Do refactored Randoop tests improve developers' effectiveness in determining the source of an issue?

To answer this question, we compared the participants' effectiveness at identifying whether the faults were in the implementation or test code. Table 5.1 summarizes the results of this investigation. The first three lines refer to results considering all classes, while the remaining present the results per class. As we can see, the tasks helped by original Randoop tests presented the better rates considering *all* (62%), *codefix* (41%), and *testfix* tasks (83%) compared with split Randoop tests (37%, 16%, and 58%) and split-renamed Randoop ones (54%, 41%, and 66%).

To our surprise, original Randoop tests performed better than refactored ones. This fact may be evidence that the refactoring, or the treated test smells, did not bring enough improvements or guidance to identify maintenance tasks correctly. However, when we compared the refactored strategies, we found that the split-renamed strategy performed better compared to the split one, which shows that automatically generated names can help developers better understand the tests.

To measure statistical significance when comparing treatments, we ran the Shapiro-Wilk [56] normality test, and the non-parametric Fisher's exact test [20] for comparisons of correctness. With a confidence of 95%, we were able statically compare strategies considering the results for *all*, *code fix*, and *testfix* tasks, individually (this analysis is similar to the one described in Section 3.3.1 and is summarized in table 5.2). With all p-values greater than 0.05, we cannot reject the null hypothesis that treatments have similar performance for *all*, *code fix*, and *testfix* tasks.

Thus, we can answer RQ7 by saying that the performed refactorings did not improve (but also not worsen) the developers' performance in determining the source of the problem. Furthermore, our results suggest that replacing bad test names has more impact on the test understanding than splitting it by assertions.

***RQ7**: The refactoring performed in the original Randoop tests, did not improve nor worsen the developers' performance in determining the source of a problem.*

Task type	**Class**	**Randoop**	**Split**	**Split-renamed**
all	all	15/24 (62%)	9/24 (37%)	13/24 (54%)
codefix	all	5/12 (41%)	2/12 (16%)	5/12 (41%)
testfix	all	10/12 (83%)	7/12 (58%)	8/12 (66%)
all	FixedOrderComparator	5/8 (62%)	3/8 (37%)	5/8 (62%)
codefix	FixedOrderComparator	1/4 (25%)	0/4 (0%)	2/4 (50%)
testfix	FixedOrderComparator	4/4 (100%)	3/4 (75%)	3/4 (75%)
all	ListPopulation	7/8 (87%)	4/8 (50%)	4/8 (58%)
codefix	ListPopulation	4/4 (100%)	2/4 (50%)	1/4 (25%)
testfix	ListPopulation	3/4 (75%)	2/4 (50%)	3/4 (75%)
all	ComparatorChain	3/8 (37%)	2/8 (25%)	4/8 (50%)
codefix	ComparatorChain	0/4 (0%)	0/4 (0%)	2/4 (50%)
testfix	ComparatorChain	3/4 (75%)	2/4 (50%)	2/4 (50%)

Table 5.1: Comparison of correct decisions given Randoop, Randoop split or Randoop split-renamed tests.

Task type	**Hiphothesys**	**p-value**	**Final ranking**
all	Randoop = Splitted	0.1482	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.7702	
	Splitted = Split-renamed	0.3852	
codefix	Randoop = Splitted	0.3707	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	1	
	Splitted = Split-renamed	0.3707	
testfix	Randoop = Splitted	0.3707	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.6404	
	Splitted = Split-renamed	1	

Table 5.2: Statistical analysis and ranking considering correct decisions for maintenance tasks.

5.3.2 RQ8: Do refactored Randoop tests improve developers' effectiveness in performing proper fixes?

To answer this question, we observed if participants were effective at producing correct fixes. Similar to our first study, a *codefix* solution is classified as correct if it does not break any additional tests from the original test suite and satisfies a manual inspection. For *test-fixing* tasks, we created reference solutions based on the original test purpose. Then, the first author ran the tests and inspected the code to classify the fix. Table 5.3 presents the collected results.

As we can see, in the tasks with split-renamed tests, developers had the better rates in *all* (50%) and *codefix* (50%). When compared with original and split tests they had the same rates (41% for *all* and 25% for *codefix*). However, for *testfix* tasks, the original Randoop and split tests had slightly better rates (58%) than split-renamed (50%). Overall, these results evidence that, while split tests have not brought any improvement to the developers' performance in producing correct fixes, split-renamed tests have shown to be better for guiding developers to the solutions. However, our statistical analysis, performed similarly to RQ7 and summarized in Table 5.4, does not allow us to conclude that the observed difference is significant.

Therefore, we can answer RQ8 by saying that, compared to original Randoop tests, developers' effectiveness in producing correct fixes was not improved when using split tests. However, when guided by split-renamed tests, developers were more effective than the other two strategies. However, this improvement is not statistically significant.

RQ8*: Developers were more effective in performing proper fixes when guided by split-renamed tests, however, this improvement was not statistically significant.*

5.3.3 RQ9: Do refactored Randoop tests improve the developers' performance to execute maintenance tasks?

For this analysis, we evaluated the time participants took to decide whether the maintenance tasks were *codefix* or *testfix*. We also measured the time spent performing the fixes.

Figure 5.2 presents the boxplots of the time spent for the participants to identify the tasks. On average, participants took 18 minutes to identify the faults. Considering the treatments, the average times were: 14 minutes for original Randoop, 17 minutes for split tests, and 22

Task type	Class	Randoop	Split	Split-renamed
all	all	10/24 (41%)	10/24 (41%)	12/24 (50%)
codefix	all	3/12 (25%)	3/12 (25%)	6/12 (50%)
testfix	all	7/12 (58%)	7/12 (58%)	6/12 (50%)
all	FixedOrderComparator	6/8 (75%)	6/8 (75%)	7/8 (87%)
codefix	FixedOrderComparator	2/4 (50%)	2/4 (50%)	3/4 (75%)
testfix	FixedOrderComparator	4/4 (100%)	4/4 (100%)	4/4 (100%)
all	ListPopulation	2/8 (25%)	1/8 (12%)	1/8 (12%)
codefix	ListPopulation	0/4 (0%)	0/4 (0%)	0/4 (0%)
testfix	ListPopulation	2/4 (50%)	1/4 (25%)	1/4 (25%)
all	ComparatorChain	2/8 (25%)	3/8 (37%)	4/8 (50%)
codefix	ComparatorChain	1/4 (25%)	1/4 (25%)	3/4 (75%)
testfix	ComparatorChain	1/4 (25%)	2/4 (50%)	1/4 (25%)

Table 5.3: Comparison of correct fixes using Randoop (original), Split or Split-renamed Randoop tests.

Task type	Hiphothesys	p-value	Final ranking
all	Randoop = Splitted	0.3376	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.0869	
	Splitted = Split-renamed	0.2438	
codefix	Randoop = Splitted	1	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.4003	
	Splitted = Split-renamed	0.4003	
testfix	Randoop = Splitted	0.0781	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.2365	
	Splitted = Split-renamed	0.6643	

Table 5.4: Statistical analysis and ranking considering correct fixes for maintenance tasks.

minutes for split-renamed tests.

Although developers took less time to identify the faults when guided by original Randoop tests, our statistical tests, following the same procedures as our first experiment and summarized in Table 5.5, did not find any significant difference among the strategies, even though the performance of developers driven by original Randoop tests is lower only on codefix tasks of the *FixedOrderComparator* class.

Figure 5.3 presents the distribution of the time that participants took to fix the faults. On average, participants took 12 minutes to fix the faults (17 minutes - original Randoop, 12 minutes - split tests, and seven minutes - split-renamed). Developers performed better in all scenarios when guided by split-renamed tests, except for the *ListPopulation* class.

Regarding the statistical tests, summarized in Table 5.6, it was not possible to notice a difference between the performances guided by split tests in relation to the other two strategies. However, the developers performed better with split-renamed tests than when guided by original Randoop tests.

Thus, we can answer RQ9 by saying that, while it is not possible to observe significant improvements in fault identification time using the refactored Randoop tests, fault fixing was less time-consuming with split-renamed tests.

***RQ9**: When guided by split-renamed tests, developers took less time in fault fixing, but the same was not observed in fault identification.*

5.3.4 RQ10: What is the developers' perception about using refactored Randoop tests in maintenance tasks?

To answer RQ10, we looked at participants' responses to the questionnaire at the end of the study. Figure 5.4 summarizes the answers for *testfix* and *codefix* tasks. In general, participants found the tasks clear (Question 1) and they had enough time to finish the tasks (Question 2). Participants found it easier to identify the fault type (Question 3) using split-renamed tests for both maintenance tasks. However, they found it easier to identify the fault type when guided by original Randoop than split tests. These results go against those presented in table 5.1, which shows that developers were more at identifying faults when guided by Randoop tests. However, our statistical analyses did not show that the strategies

Task type	**Hiphothesys**	**p-value**	**Final ranking**
all	Randoop = Splitted	0.3376	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.0869	
	Splitted = Split-renamed	0.2438	
codefix	Randoop = Splitted	0.8428	Randoop = Splitted
	Randoop = Split-renamed	0.3262	Randoop = Split-renamed
	Splitted = Split-renamed	0.0068	Splitted > Split-renamed
testfix	Randoop = Splitted	0.0781	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.2365	
	Splitted = Split-renamed	0.2438	

Table 5.5: Statistical analysis and ranking considering identification time for maintenance tasks.

Task type	**Hiphothesys**	**p-value**	**Final ranking**
all	Randoop = Splitted	0.5040	Randoop = Splitted
	Randoop = Split-renamed	0.0490	Split-renamed > Randoop
	Splitted = Split-renamed	0.0630	Splitted = Split-renamed
codefix	Randoop = Splitted	0.5040	Randoop = Splitted = Split-renamed
	Randoop = Split-renamed	0.0490	
	Splitted = Split-renamed	0.0630	
testfix	Randoop = Splitted	0.1885	Randoop = Splitted
	Randoop = Split-renamed	0.0250	Split-renamed > Randoop
	Splitted = Split-renamed	0.3400	Splitted = Split-renamed

Table 5.6: Statistical analysis and ranking considering correction time for maintenance tasks.

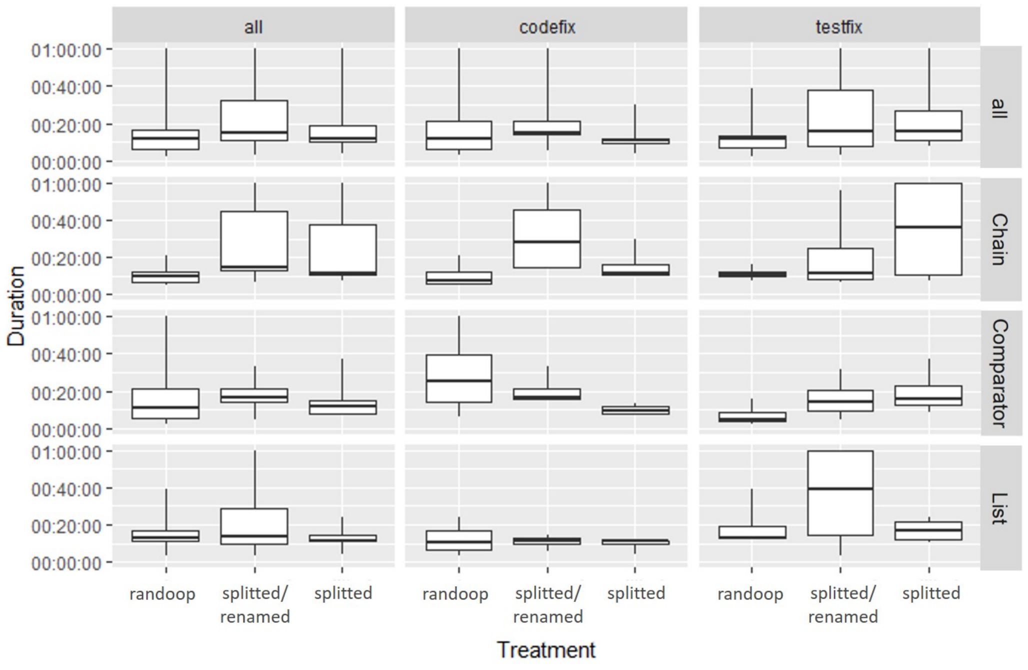


Figure 5.2: The time developers spent to identify their maintenance tasks grouped by task type and class.

have similar performance at this point.

Question 4 asked the participants' perceptions about the activity of fixing the bugs. For code fixes, the participants found it easier to perform a bug fix using refactored tests, especially the split-renamed ones. These results agree with those presented in Table 5.3, which shows that the developers were more effective in code fixes when guided by split-renamed tests. This goes according to our expectations since a less smelly test tends do be easier to follow. However, statistical tests have shown that these differences have no relevant significance.

Participants reported higher confidence about the correctness (Question 5) of their code fixes when using refactored tests, but the same was not observed for test fixes, in which participants were more confident with original Randoop tests in those scenarios. Again, these results are in accordance with those classified in Table 5.1.

Developers found that was is easier to understand the class under test (Question 6) and the tests (Question 7) when they were guided by original Randoop tests. For test fixes, they found it easier when guided by the refactored tests. In question 8, for code fix tasks, participants pointed out that the refactored tests were more useful in understanding CUT,

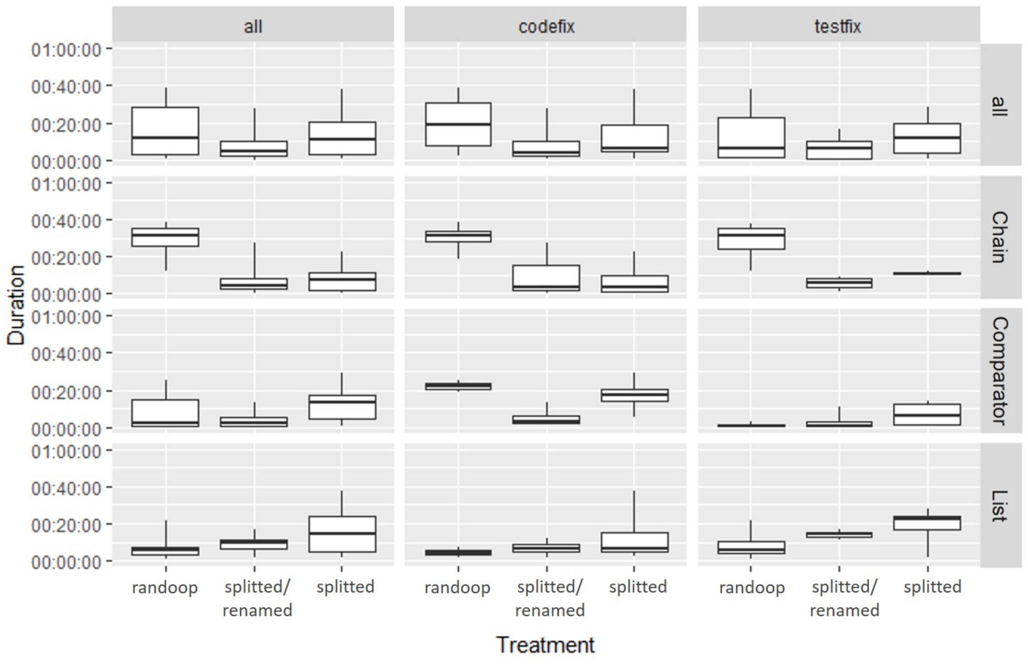


Figure 5.3: The time developers spent to fix their maintenance tasks grouped by task type and class.

especially the split-renamed tests. As for the test fixes tasks, they believe that they produced a better solution for the faults present in the Randoop tests.

In summary, we can answer RQ10 by saying that refactored tests contribute better to the understanding of CUT, facilitate the identification of maintenance activities (for both tasks), and facilitate code fixes, but developers feel more confident in their testfixes when it is done in the original Randoop tests.

***RQ10**: Refactored tests contribute more to the understanding of CUT, and facilitate the identification of maintenance activities and code fixes, but the developers feel more confident in their test fixes when it is done using original Randoop tests.*

5.3.5 Analysis by Roles

All participant roles were more effective in determining the source of a problem using the original Randoop tests, except *software engineers*, which were more effective using the refactored (renamed and split) tests (RQ7). These results may be influenced by the sample sizes. There were 17 *software engineers*, while for the other roles there were less than four participants. Regarding RQ8, *data scientists* and *software engineers* were more effective

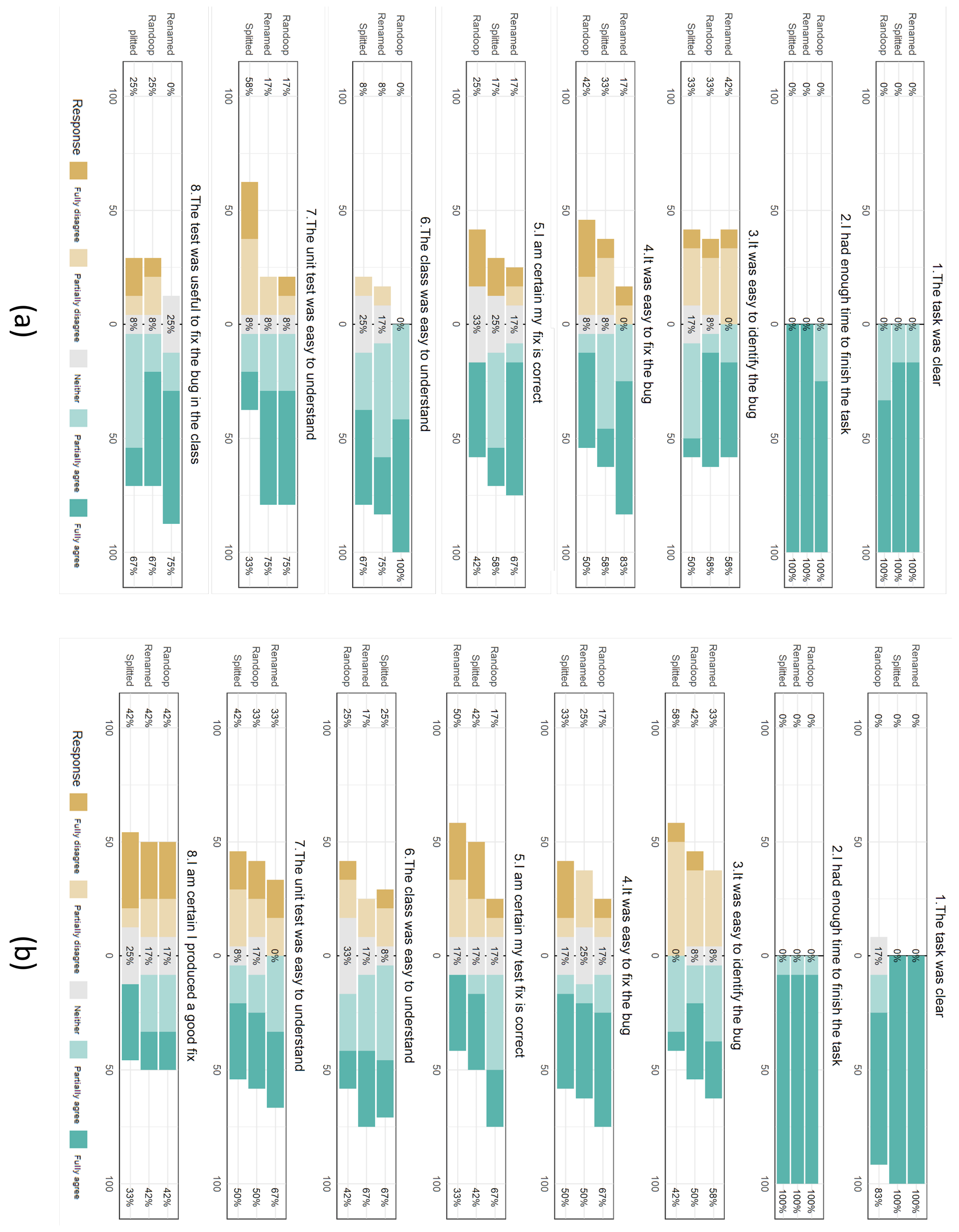


Figure 5.4: Overview of the questionnaire responses relating to (a) *codefix* and (b) *testfix* maintenance tasks.

in fixing the bug using the refactored (renamed and split) tests. On the other hand, *technical leaders* and *test analysts* were more effective using the original Randoop tests. Finally, *tech-*

nical leaders and *software engineers* took less time to identify the fault using the refactored (renamed and split) tests, while *test analysts* and *data scientists* were faster using the original Randoop tests (RQ9). Again, these results may be influenced by the sample sizes instead of the participant roles. Moreover, due to the small number of developers from each role, these results have no statistical relevance.

Similar to the survey presented in the previous chapter, we also analyzed the results only for software engineers and quality analysts. However, we also found no significant difference. The tables and graphs of this analysis can be found in Appendix B.

5.3.6 Follow-up Interviews

Here we discuss the results of our guided interviews:

- What difficulties did you experience when deciding the maintenance task to be performed?

  Most interviewees pointed out poor test code readability as their main difficulty. Non-descriptive variable names, long test statements, and confusing assert messages were mentioned as reasons for performing the maintenance task incorrectly. These test smells were not addressed by the refactored tests. Therefore, we can say that even those versions require further improvements considering other test smells;

- Do you consider the provided test case name good and representative? Did it help you identify/fix the fault?

  All interviewees pointed out the generated names (renamed refactoring) as suitable for the test cases. Some respondents stated that those tests *"guided them in solving the problem and understanding the tests"*. However, one participant stated that some of the test names were too long which ended up working as a confusing factor;

- Related to tasks with split tests: Did you consider the non-failing tests when identifying/fixing the fault?

  Most interviewees stated that they focused only on the failed test. One participant reported that he scanned the other tests but only for understanding the structure of the assertions. This shows that the refactoring strategy *Split tests* makes it easier for developers to focus on a smaller code snippet where the fault is evidenced;

- Which of the three treatments (Randoop, renamed, split-renamed tests) would you consider the best option to assist the maintenance activities and why?

Although some disadvantages were listed (e.g., duplicate and unused code, number of test cases, test suite execution time), all interviewees preferred the split-renamed option over the original Randoop tests. The main advantages listed were the opportunity to focus on a single objective (one assert per test), and test names that facilitated the understanding of what the test is doing. On the other hand, most mentioned the fact that by dividing a test per asserts some code duplication was added. When splitting a test, we use static analysis to find variable dependencies related to the asserts. To avoid compilation errors or behavioral change, any found statement related to this asserts' dependencies remained in the split tests. This may lead to some duplicate and unwanted code in the test.

5.4 Final Considerations

In this chapter, we presented our third empirical study on how developers deal with maintenance tasks now using different versions of Randoop tests (original and refactored ones). The results of this investigation were quite interesting. Although developers agree that splitting and renaming a Randoop test improves its quality (Sections 4.3 and 5.3.6), we could not find significant practical improvements when they faced a maintenance task with refactored tests. However, the use of refactored Randoop tests did not worsen the results. Moreover, the refactorings brought gains to the time spent in fixing bugs, both in the CUT and test suites, which shows that they can contribute to reducing the costs of maintenance tasks. Furthermore, the best results were obtained when removing the two test smells together (split-renamed tests), which demonstrates that the more test smells are removed, the better those tests tend to be adopted in practice. These results may guide further studies focused on improving tests automatically generated by Randoop and similar tools. In the next chapter, we present some threats related to our studies.

Chapter 6

Threats to validity

This work is based on a series of empirical studies. In this chapter, we discuss the main threats to our conclusions.

In terms of *construct validity*, both studies (Chapter 3 and 5) reuse most of the artifacts (classes and faults) from other empirical study [61]. We decided to reuse those artifacts to be able to compare results. The added extra class and fault added were also inspired by previous unit testing empirical study [58]. Although limited, those artifacts were selected from open-source projects and reflect real-world faults. Moreover, a more complex configuration (more classes and test suites with more than a single test) would be impractical. Even with such limited artifacts, participants took an average of 30 minutes to find and fix the faults per session (total of 1.5h per participant). It is important to remember that participants are real developers, which often have limited time to participate in such studies.

We also believe that the used artifacts in each maintenance task (single class and failing test case) emulate real scenarios. When identifying and fixing faults (source code or test code), developers often focus on a single class and/or small edits [33]. That said, our results do not generalize beyond our dataset of subject programs, faults, and tests. For instance, a different set of tests cases may lead to different results. However, by selecting a test that fails after fault injection, we guarantee it relates to the fault and, therefore, it can help detect and fix the fault. Thus, we believe the selected artifacts are good representatives maintenance tasks.

We did not assess the quality of the code nor selected test cases, as its not our goal. We used developers' output artifacts, their video recordings, and multiple-choice questions to investigate aspects such as effectiveness and perception. Other strategies could be used in this sense, however, our goal was to see the practical aspects of a maintenance task using failing test cases. In addition, we counterbalanced the order and task assignment to mitigate learning effects.

We adapted two refactoring strategies (Rename and Extract Method) for improving Randoop tests (Test Renaming and Split Tests). To apply them, we proposed a script that reuses and adapts Ermira et al.'s renaming strategy [18] and the Eclipse Refactoring Engine. The Eclipse Refactoring Engine was used by other works [5] and is known to have a robust test suite that validates its transformations. Moreover, to validate our implementation, a series of tests were conducted and manually validated by the authors. Although, a larger scale study is needed to ensure the use of these techniques in practice.

As for *conclusion validity*, our studies deal with a limited number of participants. Again, since we chose to work with real developers we were subjected to the availability of developers from partner companies. However, we selected participants from different projects, with different roles and levels of experience. We believe that by working with real-world developers we apply a more practical investigation. Works on empirical software engineering (e.g., [12; 35]) reinforce the need for real-world participants in empirical studies. Moreover, our study ended up providing interesting conclusions that even went against a similar study that used students as participants [61].

To mitigate *internal validity*, before the participants started their maintenance tasks, we ran a short tutorial on the procedure of the tasks. Moreover, they were familiar with the general aspects of a Java/JUnit application and identifying and fixing bugs. The participants were not familiar with the tests and CUTs before the study sessions. However, this scenario resembles a very common one in real projects, where developers need to maintain others or even legacy code. Furthermore, we cannot generalize our findings to contexts where developers maintain familiar code. Furthermore, during the sections, the first author was available for questions regarding the study procedure and provided environment.

Regarding *external validity*, our studies delt with Brazilians developers, we evaluated two test generation tools (Evosuite and Randoop), both used for unit tests in Java language, two refactoring techniques and three test smells. Due to these limitations, the found results may not be generalized.

Chapter 7

Related work

In this chapter, we relate our work to other important research. First, we discuss the methodology used to find those papers (Section 7.1). Next, we present a series of works that compared the usage of manual and generated test cases (Section 7.2), test smells on generated tests (Section 7.3), and test Code Improvement (Section 7.4). Finally, in Section 7.5, final considerations are discussed.

7.1 Methodology

We found the related work by performing ad-hoc queries (varying the search keys) to a series of online repositories: IEEE Xplore Digital Library, ACM Digital Library, and Google Scholar. For that, we considered keywords such as "test smells", "refactoring", "software testing", "automated test generation", "automatically generated unit tests", "test naming", "software maintenance", and "test maintenance". The keywords were verified both in the title and/or in the text of the paper's content of the work.

The articles that included the keywords were selected for reading, while the others were discarded. In addition, we followed the track of references to find new studies, a technique known as snowball sampling [71].

By following this process, we selected 42 studies. After filtering, reading the abstract, and doing a superficial reading in some cases, we ended up with 28 papers. They are presented below.

7.2 Comparing Manual and Generated Test Cases

Regarding comparing manual and generated test cases, there are works that are worth mentioning. Fraser et al. [25] and Rojas et al. [58] compare the behavior of participants when writing tests to the use of test generators. By not finding measurable improvements in the number of bugs actually found by developers, they confirmed the necessity of increasing the usability of automated unit test generation tools to better integrate them during software development, and to educate software developers on how to best use those tools. Alves et al. [64; 63] investigate whether generated tests (Randoop and Evosuite) can be used to find specific refactoring faults. Panichella et al. [53] run an empirical study with developers to investigate test understandability when comparing regular generated tests and generated tests with textual test summaries (comments in the test class, explaining each test case and the CUT). They concluded that developers find twice as many bugs using tests with summaries. Daka et al. [16] investigate the effect of test readability on the time developers take to predict generated test outputs. They conclude that readability has a significant impact on the time developers need to reach a decision. These results corroborate our findings since participants of our study complained that some generated tests require improvements regarding code readability.

Our empirical study was greatly inspired by Shamshiri et al.'s work [61]. The authors run a study with students to investigate how they perform maintenance tasks using Evosuite and manually-written tests. In our study, we focused on real developers and we deal with a more comprehensive set of strategies: study 01 - manual, Evosuite tool, and Randoop tool; study 02 - original and refactored versions of Randoop tests. As discussed before, in several points, our conclusions differ from Shamshiri et al.'s. Therefore, we believe both works are complementary, they apply an investigation using different contexts and treatments. Moreover, we expanded the investigation by running a survey with developers and a new study on effectiveness of generated tests, now focusing in different versions of Randoop tests cases that reduced test smells.

7.3 Test Smells on Generated Tests

There are several works about test smells in unit tests. Most of them focus on analysis of occurrence, prevalence and impacts of test smells on software project (e.g., [9; 8; 27; 66; 52]). Moreover, many studies propose tools to detect/refactoring test smells (e.g., [70; 60; 54]). However, few studies address test smells in automatically generated tests.

In their study, Palomba et. al [51] investigate the diffusion of test smells in automatically generated unit tests. Their findings indicated a high diffusion of test smells and a strong positive correlation with characteristics of structural elements. Moreover, they investigated whether CUT characteristics influence test smells generation [29]. They found that test smells generated by random algorithms (e.g., Randoop) are not influenced by CUT characteristics.

Virgilio et al. [69] compared tests generated by Randoop and Evosuite with the existing tests suite of open-source projects, regarding the presence of 19 types of test smells. The results indicate that the existing tests had a smaller distribution of test smells compared to the generated by tools, Randoop tests had a larger test smells distribution when compared to Evosuite tests.

In our work, we focused on assessing developers' perception of the use of refactored Randoop tests, using well-known refactoring to solve three kinds of test smells (Chapter 4) and how they impact on the performance of tasks maintenance (Chapter 5).

7.4 Test Code Improvements

About strategies that use refactoring-like transformations in test cases, we can discuss a series of works. J. Xuan and M. Monperrus [72] propose an approach that divides test cases with the goal of improving fault localization. Stefano et. al. [36] present DARTS, an IntelliJ plug-in that detects and refactors tests based on multiple asserts. They extract asserts to a private method that is called by original test. Zhang et al. [73] propose an approach to generate names for unit tests based on common test structures. Given a test, it identifies the action (e.g., the method being tested), the testing scenario (e.g., the parameters and context of the action), and the expected result (e.g., an assertion). The paper does not focus on generated tests, however, we believe this strategy might not work well in this context. Testing scenario identification often depends on variables with descriptive names and the expected result is assumed to be a single assertion. Moreover, generated tests often cover several methods in a single test case, which may confuse the renaming strategy.

Allamanis et al. [4] apply a log-bilinear neural network model that suggests method names based on source code features. Again, this is a technique that might be hard to apply in generated tests, since those tests tend to use short sequences of calls and less descriptive names.

Ermira Daka et al. [18] present an approach for generating Evosuite test names. It uses coverage goals to create the names. As as far as we know, this is the first approach that deals with generated tests. Therefore, we adapted it to use with Randoop tests (Section 4.2.2). Moreover, we ran a survey (Chapter 4) and an empirical study (Chapter 5) to evaluate it in this novel scenario.

Our results show that, although important, applying extract and rename refactorings might not be enough to improve its quality. Other works have proposed strategies for refactoring test cases focused on other quality aspects, such as improving identifies names, code simplification, and quality metrics. For instance, Thies and Roth [67] propose an approach based on static analysis to support the identifiers renaming. Allamanis et al. [3] proposed NATURALIZE, an approach based on an n-gram language model that suggests new names for identifiers. The n-gram model predicts the probability of the next token, considering the previous n-1 tokens. NATURALIZE learns coding conventions from the code base, promoting consistency in the use of identifiers. Lin et al. [38] evolve NATURALIZE with an approach that combines code analysis and n-gram language models.

Another way to improve the quality of unit tests is to simplify them. Search-based approaches can be used to make tests more understandable by generating more realistic scenarios [26], closer to natural language [2], or with better quality metrics (e.g., coupling and cohesion) [50]. Those strategies are yet to be evaluated when dealing with generated tests.

7.5 Final Considerations

In this chapter, we presented the related work. We discussed papers that focusing on the comparison between usage of manually-generated and generated tests, tests smells in generated tests, and test code improvements. In general, we can say that few works focus on how these issues impact software maintenance tasks. Regarding test code improvement researches, most of them were dedicated to manually-generated tests, and present limitations to be applied in generated tests. In the next chapter, we present our final conclusions and future work.

Chapter 8

Conclusions and Future Works

Developers often use failing test cases to guide maintenance tasks. However, good and trustworthy test cases are not always available. Generated suites have become an option to cope with this problem. The goal is to reduce the burden of creating sound test cases. However, we need to assess how developers perform and perceive the use of generated tests during maintenance. In this context, we ran three empirical studies with a total of 126 real developers, in various roles.

The first study compared how 20 developers perform maintenance tasks with two types of automatically generated (Randoop and Evosuite) and manually-written test cases. We found that developers were more accurate at identifying maintenance tasks when using Evosuite tests, while they were equally accurate when using manually written and Randoop tests. Moreover, they were similarly effective at producing correct bug fixes using the three strategies (manually written, Evosuite, and Randoop). Regarding their perspectives, developers were more confident that produced correct outputs when using Evosuite tests, but they found manual tests a better proxy for understading the classes under test's behavior.

Test smells may be the main factors that impair test code comprehension, readability, and maintenance. Randoop tests often include a number of test smells, such as non-descriptive names, assertion roulette, duplicate assert, eager test, lazy test, and magic number. This observation motivated us to perform the second study, a survey with 82 developers, that evaluated developers' perception about refactored Randoop tests. We found that automatic renaming is well-received. Moreover, they preferred refactored Randoop tests over original ones. They also reported needing more refactorings (e.g., better variable names) to fully accept Randoop tests.

Finally, our third study replicated the first one focusing on evaluating whether refactored Randoop tests have an impact on the performance of maintenance tests when compared to original ones. We found that refactorings did not improve the performance when identifying maintenance tasks. However, developers were more effective, and a little bit more efficient to fix the faults with refactored Randoop tests. This task was also less time-consuming.

Based on those results we can conclude that automatically generated tests, specially Evosuite's, can be a great help for identifying/fixing faults during maintenance, which differs from previous findings [61]. Randoop tests, although effective for fault identification, require improvements to be better accepted in practice. Refactoring transformations might be a good way to improve them. The suggested refactorings (split-rename) were better appreciated by developers, but have shown little improvements in developers' time spent in fixing bugs. We believe that other refactoring types could also be used in this context, such as *variables renaming* and combined *method extractions*.

In future work, we plan to extend our studies with a larger group of participants and consider other configurations (e.g., working with an entire test suite instead of a single failing test). We also intend to improve the generation tools (Evosuite and Randoop) in order to solve some of the issues that cause test smells. Moreover, we plan to investigate the use of different refactoring transformations (e.g., variable/field Renaming) and their impact on developers' performance of maintenance tasks with generated tests.

Bibliography

[1] Ieee standard for software maintenance. *IEEE Std 1219-1998*, pages 1–56, 1998.

[2] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, 2013.

[3] M. Allamanis, E. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, 2014.

[4] M. Allamanis, E. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, 2015.

[5] E. Alves, P. Machado, T. Massoni, and S. Santos. A refactoring-based approach for test case selection and prioritization. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 93–99, 2013.

[6] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24:219–250, 2014.

[7] S. Bauersfeld, T. Vos, and K. Lakhotia. Unit testing tool competitions - lessons learned. In *Future Internet Testing*, pages 75–94, 2014.

[8] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012*

28th IEEE International Conference on Software Maintenance (ICSM), pages 56–65, 2012.

[9] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20, 08 2014.

[10] K. Beck. *Test Driven Development. By Example (Addison-Wesley Signature).* 2002.

[11] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87, 2000.

[12] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, pages 239–249, 2003.

[13] IEEE Standards Coordinating Committee et al. Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos. *CA: IEEE Computer Society*, 169, 1990.

[14] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[15] W. Cunningham. Bugs in the test. *http://wiki.c2.com/?BugsInTheTests*.

[16] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118, 2015.

[17] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[18] E. Daka, J. Rojas, and G. Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT*

International Symposium on Software Testing and Analysis, ISSTA 2017, pages 57–67, 2017.

[19] A. 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 (XP2001*, pages 92–95, 2001.

[20] R. Fisher. On the interpretation of χ^2 from contingency tables, and the calculation of p, 1922.

[21] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[22] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, pages 1–42.

[23] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, 2011.

[24] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369, 2013.

[25] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology*, 24, 2015.

[26] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 80–89, 2011.

[27] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018.

[28] R. Glass. *Software Engineering: Facts and Fallacies*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[29] G Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019.

[30] M. Harrold and M. Souffa. An incremental approach to unit testing during maintenance. In *Proceedings. Conference on Software Maintenance, 1988.*, pages 362–367, 1988.

[31] W. Herculano, M. Mongiovi, and E. Alves. Manually written or generated tests? a study with developers and maintenance tasks. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES ’20, pages 273–282, 2020.

[32] T. Hesterberg. Bootstrap. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3:497–526, 2011.

[33] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[34] F. Kifetew, X. Devroey, and U. Rueda. Java unit testing tool competition - seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 15–20, 2019.

[35] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28:721–734, 2002.

[36] S. Lambiase, A. Cupito, F. Pecorelli, A. De Lucia, and F. Palomba. Just-in-time test smell detection and refactoring: The darts project. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC ’20, pages 441–445, 2020.

[37] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., USA, 1980.

[38] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 81–90, 2017.

[39] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[40] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. 2006.

[41] M. Mongiovi. Scaling testing of refactoring engines. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 674–676, 2016.

[42] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44:429–452, 2018.

[43] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 252–266, 2007.

[44] G. Myers, C. Sandler, and T. Badgett. *The art of software testing*. Wiley Publishing, 3rd edition, 2012.

[45] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41:81–86, 1998.

[46] Int. Standards Organisation. Iso12207 information technology - software life cycle processes. 1995.

[47] C. Pacheco and M. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, 2007.

[48] C. Pacheco, S. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 87–96, 2008.

[49] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, 2007.

[50] 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 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 130–141, 2016.

[51] F. Palomba and A. Zaidman. Notice of retraction: Does refactoring of test smells induce fixing flaky tests? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12, 2017.

[52] F. Palomba and A. Zaidman. The smell of fear: on the relation between test smells and flaky tests. *Empirical Software Engineering*, 24:2907–2946, 2019.

[53] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558, 2016.

[54] Anthony Peruma, Khalid Almalki, Christian Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. pages 1650–1654, 11 2020.

[55] Rudolf Ramler, Dietmar Winkler, and Martina Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? pages 286–293, 2012.

[56] N. Razali and B. Yap. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of Statistical Modeling Analytics*, 2:21–33, 2011.

[57] S. Reichhart, T. Gîrba, and S. Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6:231–251, 2007.

[58] J. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 338–349, 2015.

[59] U. Rueda, F. Kifetew, and A. Panichella. Java unit testing tool competition - sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pages 22–29, 2018.

[60] Railana Santana, Luana Martins, Larissa Soares, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. Raide: a tool for assertion roulette and duplicate assert identification and refactoring. pages 374–379, 10 2020.

[61] S. Shamshiri, J. Rojas, J. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018.

[62] Sina Shamshiri, Rene Just, José Miguel Rojas, Gordon Fraser, Phil Mcminn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. 2015.

[63] I. Silva, E. Alves, and W. Andrade. Analyzing automatic test generation tools for refactoring validation. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*, pages 38–44, 2017.

[64] I. Silva, E. Alves, and P. Machado. Can automated test case generation cope with extract method validation? In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, pages 152–161, 2018.

[65] G. Soares. Making program refactoring safer. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 521–522, 2010.

[66] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12, 2018.

[67] Andreas Thies and Christian Roth. Recommending rename refactorings. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 1–5, 2010.

[68] M. Umar. Comprehensive study of software testing: Categories, levels, techniques, and types. *International Journal of Advance Research, Ideas and Innovations in Technology*, 5:32–40, 2019.

[69] T. Virgínio, L. Martins, L. Soares, R. Santana, H. Costa, and I. Machado. An empirical study of automatically-generated tests from the perspective of test smells. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, pages 92–96, 2020.

[70] Tássio Virgínio, Luana Martins, Larissa Soares, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. Jnose: Java test smell detector. pages 564–569, 10 2020.

[71] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, 2014.

[72] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, 2014.

[73] B. Zhang, E. Hill, and J. Clause. Towards automatically generating descriptive names for unit tests. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 625–636.

Appendix A

Experiment procedure

In this appendix, we present a more detailed overview of the procedure in the studies depicted in Chapters 3 and 5. They both follow the same procedure, but the first was run in person while the second was run remotely, due to the Covid-19 pandemic.

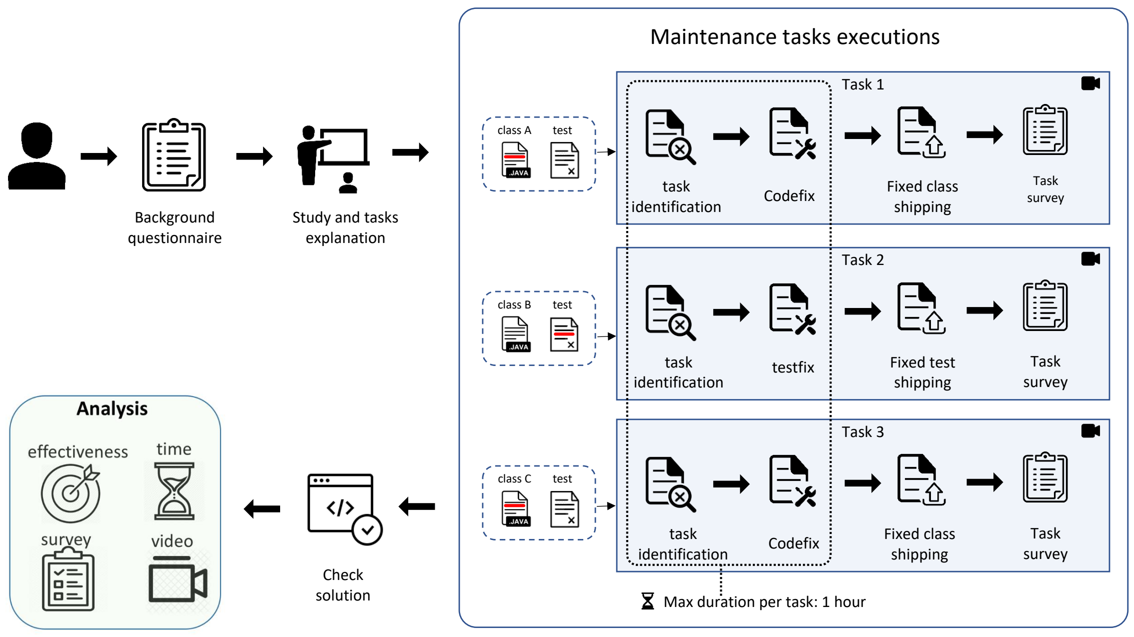


Figure A.1: Detailed procedure overview of the studies from chapters 3 and 5, considering a single participant

Appendix B

Extra analysis

In this appendix, we present tables and graphs from the survey results and the study with refactored Randoop tests (Chapters 4 and 5, respectively). Here, we filter the results by considering only participants that usually deal with unit tests in practice (e.g., software engineers or quality analysts).

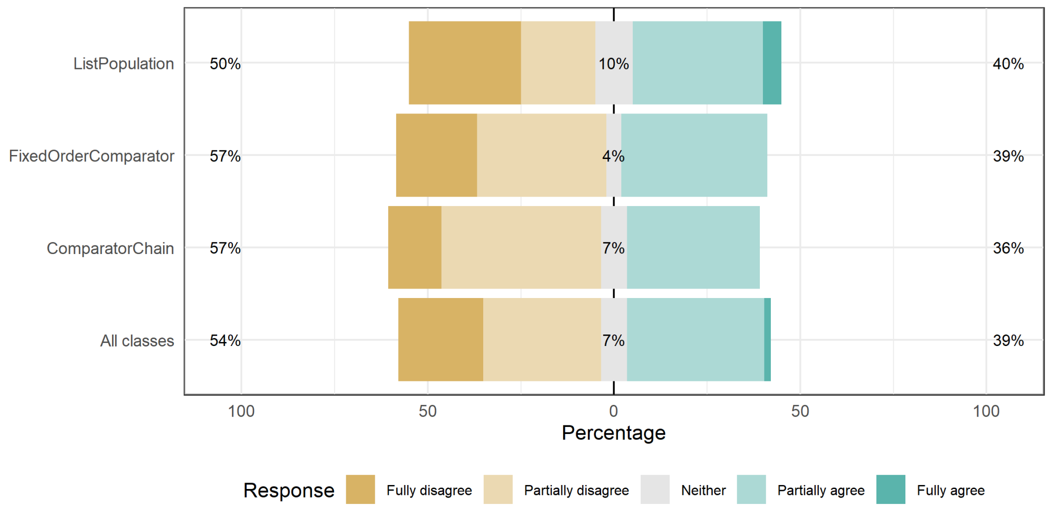


Figure B.1: Overview of survey answers about the agreement with automatically generated test names, considering only software engineers and quality analysts participants.

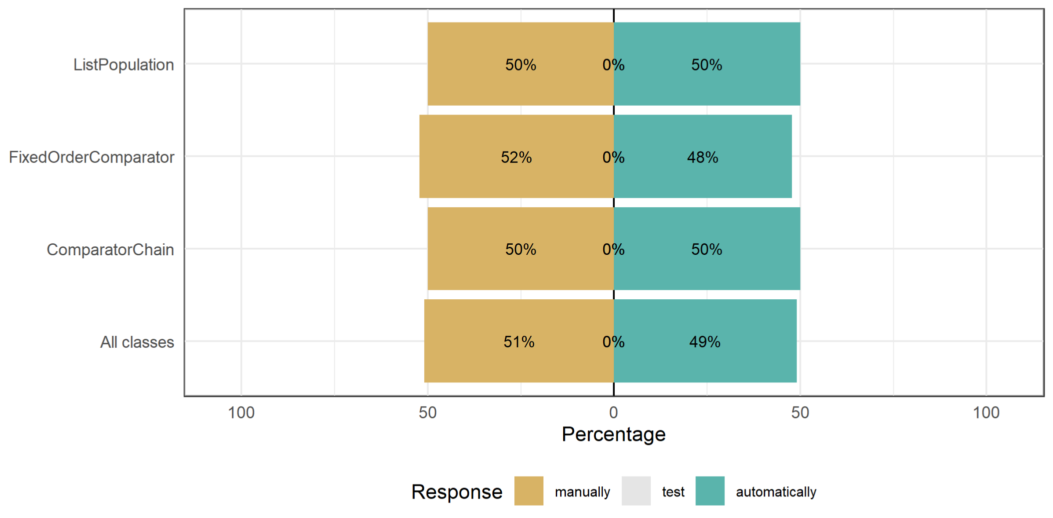


Figure B.2: Overview of survey answers about more appropriate test names, considering only software engineers and quality analysts participants.

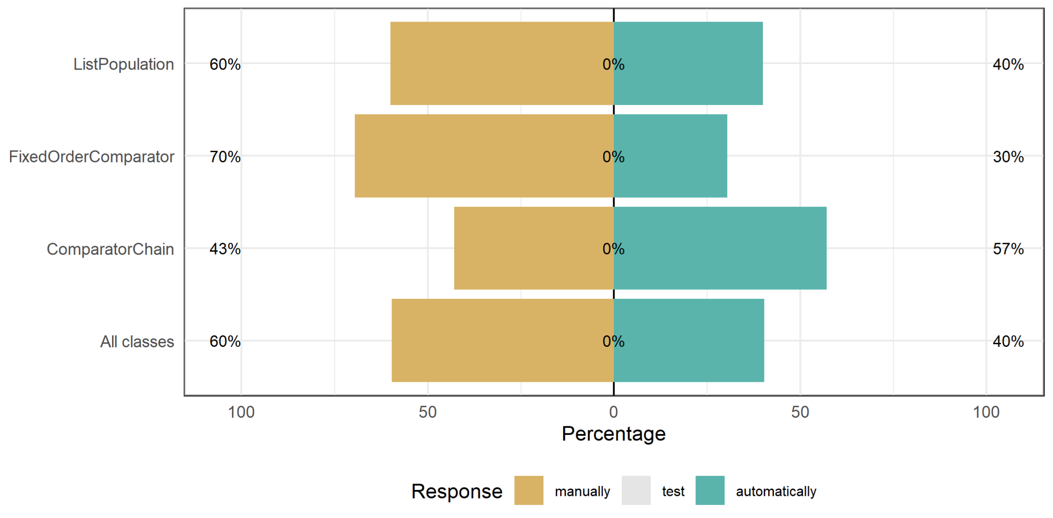


Figure B.3: Survey answers about the test that should exercise a CUT, considering only software engineers and quality analysts participants.

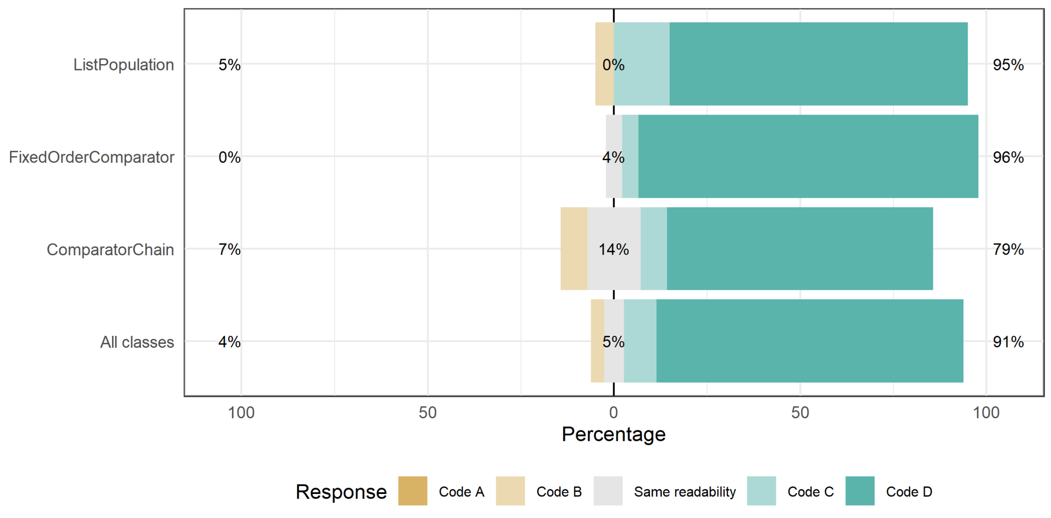


Figure B.4: Survey answers about the most readable test code, considering only software engineers and quality analysts participants

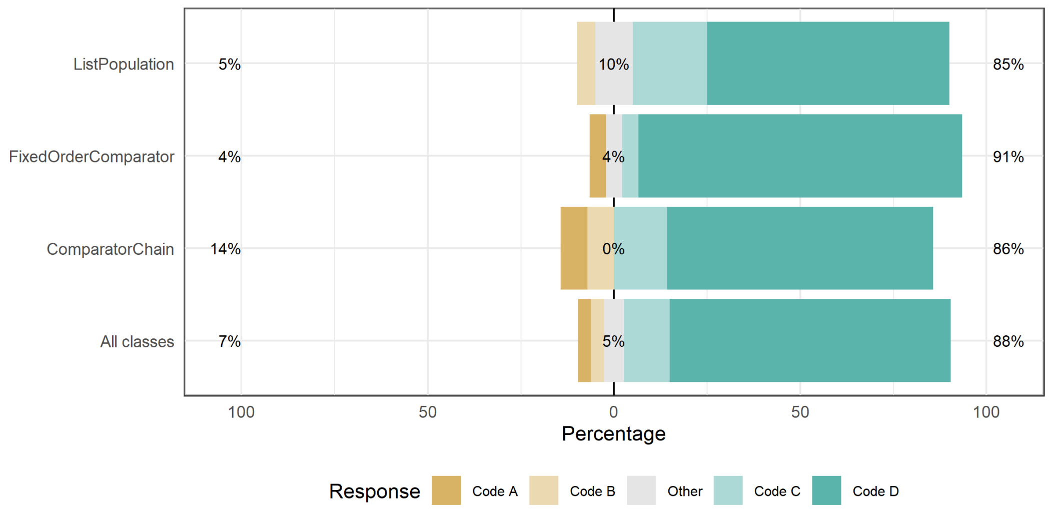


Figure B.5: Survey answers about the most appropriate test to include in a test suite, considering only software engineers and quality analysts participants

Task type	Class	Randoop	Split	Split-renamed
all	all	10/19 (42%)	10/19 (47%)	10/19 (54%)
codefix	all	4/11 (36%)	4/9 (44%)	2/8 (25%)
testfix	all	6/8 (75%)	6/10 (60%)	7/11 (63%)

Table B.1: Comparison of correct decisions given Randoop, Randoop split or Randoop split-renamed tests, considering only software engineers and quality analysts participants.

Task type	Class	Randoop	Split	Split-renamed
all	all	8/19 (42%)	9/19 (47%)	10/19 (52%)
codefix	all	3/11 (27%)	5/9 (55%)	3/8 (37%)
testfix	all	5/8 (62%)	5/10 (50%)	6/11 (54%)

Table B.2: Comparison of correct fixes using Randoop (original), Split or Split-renamed Randoop tests, considering only software engineers and quality analysts participants.

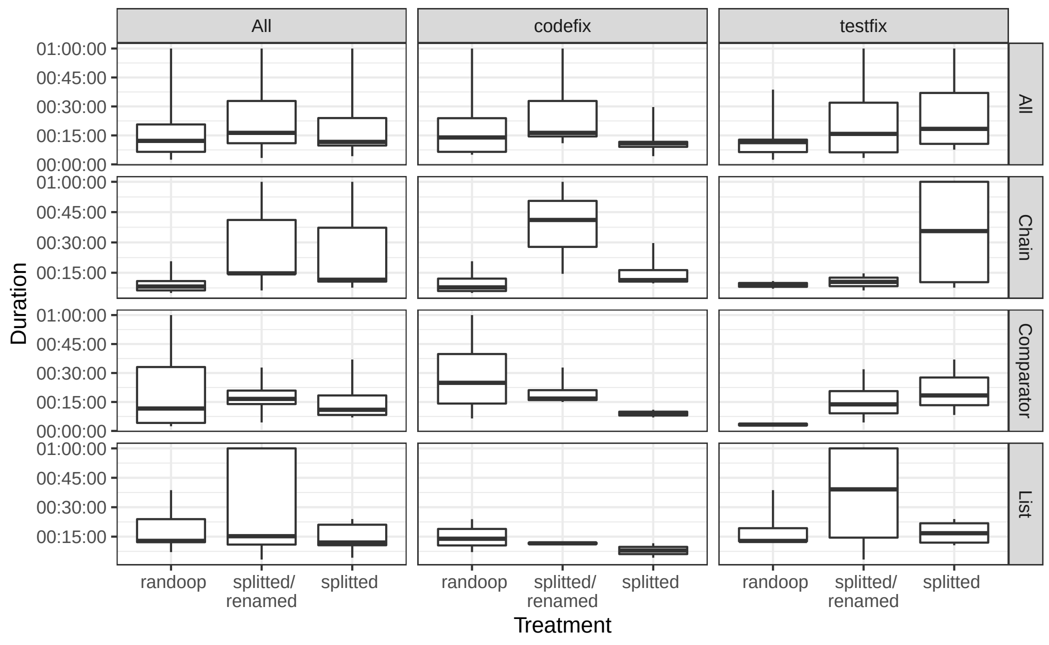


Figure B.6: The time developers spent to identify their maintenance tasks grouped by task type and class, considering only software engineers and quality analysts participants.

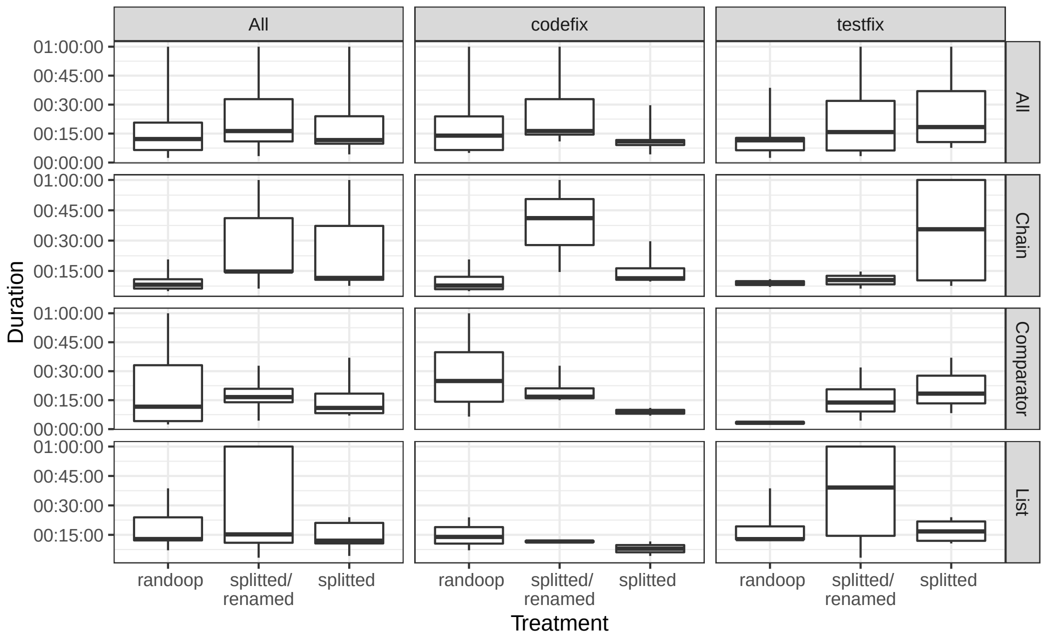


Figure B.7: The time developers spent to fix their maintenance tasks grouped by task type and class, considering only software engineers and quality analysts participants.