Smart prediction for test smell refactorings

Luana Almeida Martins

Tese de Doutorado

Universidade Federal da Bahia

Programa de Pós-Graduação em Ciência da Computação

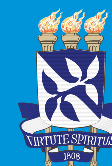
Março | 2024

DSC | 49 | 2024

Smart prediction for test smell refactorings

Luana Almeida Martins

UFBA



PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

https://pgcomp.ufba.br
pgcomp@ufba.br

Test smells are considered bad practices for developing the test code. Their presence can reduce the test code quality, thus harming software testing and maintenance activities. Software refactoring has been a key practice to handle smells and improve software quality without changing its behavior. However, existing refactoring tools target production code with very different characteristics than test code. Despite the research invested in test smell refactoring, little is known about whether current refactorings improve the test code quality. In this thesis, a machine learning-based approach is presented that can help developers decide when and how to refactor test smells. First, we aim to mine refactorings performed by developers to derive a catalog of test-specific refactorings and their impact on the test code. Our findings show that developers prefer specific features of the testing frameworks, which may lead to test smells such as Inappropriate Assertion and Exception Handling. While the refactorings proposed in the literature aligned with the evolution of testing frameworks to help refactor test smells, the Inappropriate Assertion remains unexplored in the literature. Second, we aim to understand whether developers target low-quality test codes to perform refactorings and the effects of refactorings on test code quality improvement. Our findings show that low-quality test code, especially regarding structural metrics, is more likely to undergo refactorings. Common refactorings between test and production code contribute more to improving test code quality in terms of cohesion, size, and complexity. Test-specific refactorings enhance quality concerning the resolution of test smells. Third, we aim to learn whether developers would perform refactorings and which refactorings they would apply to improve the test code quality. Results indicate that the accuracy of Support Vector Machines models varies between 30% and 100% in different projects for detecting when a developer would perform a refactoring. However, accuracy decreases for detecting specific refactorings due to the low data on test refactorings found in analyzed projects. Overall, this research demonstrates the feasibility of using structural metrics and test smells for detecting test refactorings. In addition, it highlights the need for improvements through the analysis of synthetic data and project development context. The proposed approach supports the detection and refactoring of test smells aligned with development practices currently adopted by developers.

Palavras-chave: Test refactoring. Test smells. Machine Learning.

UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SMART PREDICTION FOR TEST SMELL REFACTORINGS

LUANA ALMEIDA MARTINS

TESE DE DOUTORADO

Salvador
26 de Março de 2024

LUANA ALMEIDA MARTINS

SMART PREDICTION FOR TEST SMELL REFACTORINGS

Esta Qualificação de Doutorado foi apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Ivan do Carmo Machado
Co-orientador: Prof. Dr. Heitor Augustus Xavier Costa

Salvador
26 de Março de 2024

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias
Prof. Omar Catunda, SIBI – UFBA.

M386 Martins, Luana Almeida.
Smart prediction for test smell refactorings / Luana Almeida Martins – Salvador, 2024.
165p.: il.

Orientador: Prof. Dr. Ivan do Carmo Machado.
Co-orientador: Prof. Dr. Heitor Augustus Xavier Costa.
Tese (Doutorado) – Universidade Federal da Bahia, Instituto de Computação, 2024.

1. Computação. 2. Test Smells. 3. Máquina - Aprendizado. I. Ivan do Carmo Machado. II. Heitor Augustus Xavier Costa. III. Universidade Federal da Bahia. Instituto de Computação. IV. Título.

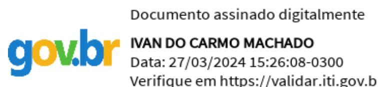
CDU – 004.412.2

Termo de Aprovação

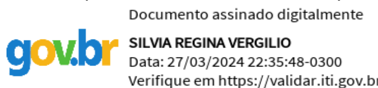
Luana Almeida Martins

Smart prediction for test smells refactorings

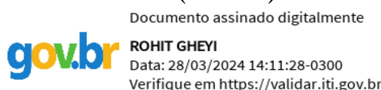
Esta tese foi julgada adequada à obtenção do título de Doutora em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da UFBA.



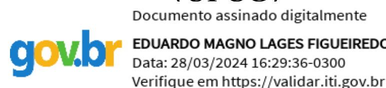
Prof. Dr. Ivan do Carmo Machado
(Orientador -UFBA)



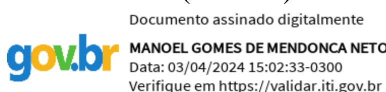
Profa. Dra. Silvia Regina Vergilio
(UFPR)



Prof. Dr. Rohit Gheyi
(UFCG)



Prof. Dr. Eduardo Magno Lages Figueiredo
(UFMG)



Prof. Dr. Manoel Gomes de Mendonça Neto
(UFBA)

This thesis is dedicated to my parents,
Jair and Marilda.

ACKNOWLEDGEMENTS

I would like to express my gratitude to a number of people who supported me throughout my PhD journey. Firstly, I am deeply thankful to my supervisor, Professor Ivan Machado, for his invaluable guidance, encouragement, and support that have significantly contributed to my academic achievements. I would also like to express my appreciation to my co-supervisor, Professor Heitor Costa, for his guidance and support during the entirety of my research journey. I would also like to thank Professor Fabio Palomba, who supervised me during my internship at Università degli Studi di Salerno and provided me with feedback during the later studies aggregated into this thesis.

I thank my colleagues from the Applied Research in Software Engineering (Aries) Lab and the Laboratory of Applied Artificial Intelligence (LIAA) at the Federal University of Bahia for our lively discussions whenever we get together. In particular, I want to thank Denivan Campos, Railana Santana, Renata Souza, Tassio Virginio, and Joselito Mota. I would like also to thank my colleagues from the Software Engineering in Salerno (SeSa) Lab, whom I shared space with during my internship. In particular, I thank Gilberto Recupito, Valeria Pontillo, and Giusy Anusiata for their dedication to helping ensure a smooth transition during my stay in Salerno.

I am grateful for the financial support I received from Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) grant BOL0188/2020 and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for the scholarship during the internship in Salerno. I am also grateful for the ACM-W scholarship and ACM SIGSOFT travel grants, which allowed me to attend two editions of the International Conference in Software Engineering (ICSE 2022 and 2023) in person. During the conferences and internships, I met mentors who are experts in my research area and found potential collaborators to work with. In particular, I thank Taher Ghaleb and Valeria Pontillo for working with me in different stages of my research.

Finally, I am deeply thankful to my family for their moral support, assistance, and understanding throughout my Ph.D. journey.

RESUMO

Os test smells são considerados más práticas durante o desenvolvimento do código de teste. Sua presença pode reduzir a qualidade do código de teste, prejudicando as atividades de teste e manutenção de software. A refatoração de software é uma prática fundamental para lidar com smells e melhorar a qualidade do software sem alterar seu comportamento. No entanto, as ferramentas de refatoração existentes são voltadas para o código de produção, com características muito diferentes do código de teste. Apesar do esforço da comunidade em investigar sobre refatorações de test smells, pouco se sabe sobre os efeitos das refatorações para a qualidade do código de teste. Nesta tese, é apresentada uma abordagem baseada em aprendizado de máquina que pode ajudar os desenvolvedores a decidir quando e como refatorar os test smells. O primeiro objetivo é minerar as refatorações realizadas por desenvolvedores a fim de derivar um catálogo de refatorações de teste e seu impacto no código de teste. Como resultados, pôde-se perceber que os desenvolvedores têm preferência por recursos específicos dos frameworks de teste, o que pode levar a test smells como o Inappropriate Assertion e o Exception Handling. Enquanto as refatorações propostas na literatura alinhadas com a evolução dos frameworks de teste auxiliam na refatoração de test smells, o Inappropriate Assertion permanece pouco explorado na literatura. O segundo objetivo busca entender se os códigos de teste com baixa qualidade são alvos de refatoração pelos desenvolvedores e os efeitos das refatorações para a melhoria da qualidade. Como resultados, pôde-se observar que códigos de teste com baixa qualidade, em especial, em termos de métricas estruturais, possuem mais possibilidade de sofrer refatorações. Além disso, as refactorações comuns entre código de teste e produção ajudam a melhorar a qualidade do código de teste em relação à coesão, tamanho e complexidade, enquanto que, refatorações específicas de teste ajudam na melhoria da qualidade em relação a resolução de test smells. O terceiro objetivo utiliza aprendizado de máquina para classificar onde e como os desenvolvedores realizam refatorações teste com o potencial de corrigir os test smells. Os resultados indicam que a acurácia do aprendizado de máquina, utilizando o algoritmo Support Vector Machines, varia entre 30% e 100% em diferentes projetos para a detecção de quando o desenvolvedor realizaria alguma refatoração no código. Porém, acurácia diminui para a detecção de refatorações específicas, devido à quantidade de refatorações encontradas nos projetos analisados. De modo geral, esta pesquisa mostra a viabilidade do uso de métricas e test smells para a detecção de refatorações de teste, evidenciando ainda a necessidade de melhorias por meio da análise de dados sintéticos e do contexto de desenvolvimento dos projetos. A abordagem apoia a detecção e refatoração de test smells alinhadas às práticas de desenvolvimento atualmente adotadas pelos desenvolvedores.

Palavras-chave: Refatoração de teste. Test smells. Aprendizado de máquina.

ABSTRACT

Test smells are considered bad practices for developing the test code. Their presence can reduce the test code quality, thus harming software testing and maintenance activities. Software refactoring has been a key practice to handle smells and improve software quality without changing its behavior. However, existing refactoring tools target production code with very different characteristics than test code. Despite the research invested in test smell refactoring, little is known about whether current refactorings improve the test code quality. In this thesis, a machine learning-based approach is presented that can help developers decide when and how to refactor test smells. First, we aim to mine refactorings performed by developers to derive a catalog of test-specific refactorings and their impact on the test code. Our findings show that developers prefer specific features of the testing frameworks, which may lead to test smells such as Inappropriate Assertion and Exception Handling. While the refactorings proposed in the literature aligned with the evolution of testing frameworks to help refactor test smells, the Inappropriate Assertion remains unexplored in the literature. Second, we aim to understand whether developers target low-quality test codes to perform refactorings and the effects of refactorings on test code quality improvement. Our findings show that low-quality test code, especially regarding structural metrics, is more likely to undergo refactorings. Common refactorings between test and production code contribute more to improving test code quality in terms of cohesion, size, and complexity. Test-specific refactorings enhance quality concerning the resolution of test smells. Third, we aim to learn whether developers would perform refactorings and which refactorings they would apply to improve the test code quality. Results indicate that the accuracy of Support Vector Machines models varies between 30% and 100% in different projects for detecting when a developer would perform a refactoring. However, accuracy decreases for detecting specific refactorings due to the low data on test refactorings found in analyzed projects. Overall, this research demonstrates the feasibility of using structural metrics and test smells for detecting test refactorings. In addition, it highlights the need for improvements through the analysis of synthetic data and project development context. The proposed approach supports the detection and refactoring of test smells aligned with development practices currently adopted by developers.

Keywords: Test refactoring. Test smells. Machine Learning.

CONTENTS

List of Figures xii

List of Tables xiv

List of Acronyms xv

Chapter 1—Introduction 1

1.1 Context and Motivation 2

1.2 Research Statement 6

1.3 Research Method 7

1.4 Overview of the proposed solution by ARIES Lab 10

1.5 Thesis outline 12

Chapter 2—Background 14

2.1 An overview of automated software testing 15

2.2 Test smells 17

2.2.1 Test smells definition and examples 17

2.2.2 Approaches to handle test smells 25

2.3 Software Refactoring 28

2.3.1 Refactoring operations 30

2.3.2 Refactoring approaches 32

2.4 Test Code Quality Assessment 33
2.5 Machine Learning 34
2.5.1 Supervised Machine Learning algorithms 34
2.5.2 Evaluation Metrics 47
2.5.3 Datasets for detection and refactoring of test smells 49
2.6 Chapter Summary 50

Chapter 3—Related work **51**

3.1 Catalogs and reviews on test smells and refactorings 52
3.2 Investigation of test smells effects on software quality 53
3.3 Investigation of test smells in different frameworks 55
3.4 Automated tools to handle test smells 56
3.5 Developers' perception and awareness of test smells 58
3.6 Refactorings to fix test smells 60
3.7 Machine Learning techniques to handle test smells 61
3.8 Limitations of prior work 62
3.9 Chapter Summary 63

Chapter 4—Mining test refactorings in practice **65**

4.1 Research Questions and Objectives 66
4.2 Approach to derive a catalog of test refactorings 67
4.2.1 Selecting subject systems 67
4.2.2 Extracting test code changes 68
4.2.3 Classifying test code changes 69

4.3 Deriving a catalog of test-specific refactorings . . . 71
4.3.1 The Exception Handling test smell . . . 73
4.3.2 The Inappropriate Assertion test smell . . . 78
4.3.3 The Assertion Roulette test smell . . . 83
4.3.4 The Bad Naming test smell . . . 86
4.3.5 Other test smells . . . 87
4.4 A Comparative and Practical Analysis of the catalog . . . 92
4.4.1 How our catalog of test smell refactorings compares to the state-of-the-art? 93
4.4.2 How our catalog of test smell refactorings is acceptable in practice? . . . 95
4.4.3 Implications for software engineering research and practice . . . 98
4.5 Threats to Validity . . . 99
4.6 Chapter Summary . . . 100

Chapter 5—How Test Refactorings Affect Test Code Quality 101

5.1 Research Questions and Objectives . . . 102
5.2 Experimental Design . . . 104
5.2.1 Context of the study . . . 104
5.2.2 Data Collection . . . 107
5.2.3 Data Analysis . . . 108
5.3 Results . . . 110
5.3.1 Are test classes performed in classes with low quality? . . . 112
5.3.2 What are the effects of test refactorings? . . . 113
5.4 Discussion . . . 118
5.5 Threats to validity . . . 119
5.6 Chapter Summary . . . 120

Chapter 6—Developer-Oriented Test Refactoring Recommendations 122

6.1 Research Questions and Objectives . . . 123
6.2 Experimental Design . . . 124
6.2.1 Context of the Study . . . 124
6.2.2 Dependent Variables . . . 125
6.2.3 Independent Variables . . . 126
6.2.4 Research Method . . . 127
6.3 Analysis of the Results . . . 130
6.3.1 Classifying Where Developers Would do Test Refactoring . . . 130
6.3.2 Classifying Specific Test Refactoring Operations . . . 133
6.4 Discussion . . . 136
6.4.1 Relation with Previous Work . . . 136
6.4.2 On the Features and their Predictive Power . . . 137
6.4.3 ML Models for Test Refactoring Recommendations: How Far Can We Go? 138
6.5 Threats To Validity . . . 139
6.6 Chapter summary . . . 140

Chapter 7—Conclusions 141

7.1 Results addressing our Goal and Research Questions . . . 141
7.2 Contributions . . . 144
7.2.1 Research Contribution . . . 144
7.2.2 Materials and Tools . . . 144
7.2.3 Academic contributions . . . 146
7.3 Future research directions . . . 147

References 149

LIST OF FIGURES

1.1 Research Method. 8
1.2 Overview of our approach. 9
1.3 Overview of the ARIES Lab framework. 11
1.4 Schematic overview of the thesis structure. 13

2.1 Garousi and Küçük (2018)'s analyzed test smells at test method level. . . 18
2.2 Lifecycle of test smells (adapted from Garousi and Küçük (2018)). 26
2.3 Building the Decision Tree for the Test Refactoring classification. The circles highlighted in blue represent the steps to classify the new instance ReceiverEmailTest. 37
2.4 Building the Trees with Random Forest algorithm for the Test Refactoring classification. 39
2.5 Building the Trees with ExT algorithm for the Test Refactoring classification. 40
2.6 Classification of the new instance *EmailReceiverTest* using LR algorithm. 42
2.7 Comparison among difference kernel functions for SVM algorithm. The x highlighted in red represent the new instance ReceiverEmailTest. 46
2.8 Boundaries for ML algorithms based on a binary dummy dataset, where the scattered points represent the training data points and respective classification. 47

4.1 Overview of the Proposed Approach. 67
4.2 Detection and refactoring the ECT test smell. 74
4.3 Detection and refactoring the InA test smell. 79
4.4 Detection and refactoring the AR test smell. 84
4.5 Detection and refactoring the BaN test smell. 86
4.6 Detection and refactoring the other test smell. 88
4.7 Comparison between our findings and the state-of-the-art regarding test smells and test refactorings. 95

5.1 Overview of the experimental design. 105
5.2 Boxplots for the distributions of metrics and test smells in the dataset. . 111

6.1 The results of the Nemenyi rank applied to the best model with balancing techniques. 131
6.2 Predictive power of Test Code and Process Metrics. 131
6.3 Predictive power of Test Smells. 132

LIST OF TABLES

2.1 Characteristics of detection and refactoring tools to handle test smells (extended from Aljedaani *et al.* (2021)). 29
2.2 Description of quality metrics (PECORELLI *et al.*, 2020b) 33
2.3 Description of process metrics (SPADINI; ANICHE; BACCHELLI, 2018) 34
2.4 Dummy dataset for test class refactorings. 35
2.5 Confusion Matrix for smell detection outcomes. 48

4.1 Characterization of the studied projects in terms of the number of commits and selected test files 69
4.2 Summary of detection and refactorings for test smells 72

5.1 Results for the statistical model considering the quality metrics (Qm_i). . 111
5.2 Results for the statistical model considering the test smells (Ts_i). 114
5.3 Results of decrease and increase of quality and Wilcoxon Rank Sum Test for Metrics (Qm_i) 115
5.4 Results of decrease and increase of quality and Wilcoxon Rank Sum Test for Metrics (Ts_i) 117

6.1 Overview of the selected JAVA open-source projects. 125
6.2 Performance of the best classifiers for each project analyzed. 133
6.3 Performance obtained to classify specific refactoring operations in QUESTDB. 134
6.4 Performance obtained when classifying the remaining seven refactoring operations. 135

LIST OF ACRONYMS

General acronyms

AST Abstract Syntax Tree
FAIR Findable, Accessible, Interoperable, and Reusable
JMX Java Management Extensions
RQ Research Questions
SSVT Setup, Stimulate, Verify, and Teardown
TTCN-3 Testing and Test Control Notation Version 3

Test smells acronyms

AR Assertion Roulette
BaN Bad Naming
CTL Conditional Test Logic
CI Constructor Initialization
DT Default Test
DpT Dependent Test
DA Duplicate Assert
ET Eager Test
EpT Empty Test
ECT Exception Handling
GF General Fixture
IgT Ignored Test
IdT Indirect Testing
InA Inappropriate Assertion
LT Lazy Test
MNT Magic Number Test
MG Mystery Guest
RP Redundant Print

RA Redundant Assertion
RO Resource Optimism
SE Sensitive Equality
ST Sleepy Test
VT Verbose Test
UT Unknown Test

Process Metrics acronyms

CCM Code Churn Max
CCA Code Churn Average
Co Commits Count
Con Contributors Count
MCon Minor Contributors Count
ConE Contributors Experience
ALC Lines Added Count
ALM Lines Added Max
ALA Lines Added Average
RLC Lines Removed Count
RLM Lines Removed Max
RLA Lines Removed Average

Code Metrics acronyms

LOC Lines of Code
NOM Number of Methods
WMC Weighted Method per Class
RFC Response for a Class
AD Assertion Density

Machine Learning acronyms

ML Machine Learning
DT Decision Trees
ExT Extra-Tree
LR Logistic Regression

MCC	Matthews Correlation Coefficient
NB	Naive Bayes
RF	Random Forest
SVM	Support Vector Machine

Chapter 1

INTRODUCTION

Software testing is a fundamental activity for software quality assurance and consists of developing helpful test cases to find bugs caused by code changes (GAROUSI; KüçüK, 2018; BELL *et al.*, 2018). However, software companies often do not prioritize software testing activities because they face difficulties selecting qualified personnel and allocating resources (MELO *et al.*, 2020; MARTINS *et al.*, 2021b). Additionally, the testing activities, when carried out manually, consume much time and effort. The effort to perform testing activities depends on the artifacts under test, e.g., large and complex production classes require more effort to develop the test codes (TERRAGNI; SALZA; PEZZE, 2020).

The complexity of the software under test, the lack of expertise, and time pressure can lead software developers to use bad practices to either design or implement the test code, resulting in the so-called test smells (DEURSEN *et al.*, 2001; TUFANO *et al.*, 2016; Silva Junior *et al.*, 2020). The presence of test smells can negatively affect the test code quality, harming the software testing and maintenance activities (BAVOTA *et al.*, 2015; PALOMBA *et al.*, 2016; SPADINI *et al.*, 2020; CAMPOS; ROCHA; MACHADO, 2021). Test smells differ from code smells defined by Fowler (1999) for the production code. Specifically, the test smells reflect problems in the test cases' organization, implementation, and interaction with one another (DEURSEN *et al.*, 2001). Therefore, test smells can require test code refactoring operations that differ from the ones applied in the production code (DEURSEN *et al.*, 2001; SOARES *et al.*, 2020).

To fix test smells, developers should refactor the test code in a way that does not change the test logic (FOWLER, 1999). While most refactoring recommendation tools

target the production code, little attention has been devoted to detecting test smells and refactoring the test code (ALJEDAANI *et al.*, 2021). Prior test code detection and refactoring strategies, based on rules and metrics, were simplistic and not derived from common practices used by developers (PERUMA *et al.*, 2022). In particular, those strategies mostly rely on predefined thresholds or rules to detect test smells. Still, deciding whether a refactoring operation fixes a particular test smell requires developers' expertise and intuition (SPADINI *et al.*, 2020).

In this work, we investigated Machine Learning (ML) techniques to classify the developers' intention to apply test refactoring and which test-specific refactoring operations they would apply. In this chapter, we contextualize the focus of this work. Section 1.1 presents our motivation and states the research problem. Section 1.2 provides details of the thesis statement, highlighting the research goals. Section 1.3 presents the steps to conduct this work. Section 1.4 presents the scope definition. Section 1.5 outlines the thesis structure.

1.1 CONTEXT AND MOTIVATION

Refactoring is modifying the software code without changing its behavior (FOWLER, 1999). The concept of behavior is different for production and test codes. The production code behavior refers to the effects of its execution, e.g., return values, changes in variables, and impacts on external resources. After refactoring the production code, the test code checks whether the behavior of the refactored code has not changed (FOWLER, 1999; DEURSEN *et al.*, 2001; BECK, 2003). Differently, the test code behavior refers to the verification in the production code. A verification comprises one assertion and actions to check the production code.

In light of the above, refactoring test code differs from the production code. Test code has test smells that reflect problems in its organization, implementation, and interaction with other codes (DEURSEN *et al.*, 2001). Refactoring of test code combines refactoring operations from Fowler (1999) and a set of test refactoring operations to deal with the particularities in the test code structure (DEURSEN *et al.*, 2001).

As a motivating example, we can consider a recurrent issue highlighted by researchers involving the evolution of structures used in JUnit over time to handle expected exceptions in test cases (SOARES *et al.*, 2022). Listing 1.1 shows the `testErrorListener`

```
71 @Test
72 public void testErrorListener() throws Exception {
73     try {
74         RouteBuilder builder = createRouteBuilder();
75         CamelContext context = new DefaultCamelContext();
76         context.getRegistry().bind("myListener", listener);
77         context.addRoutes(builder);
78         context.start();
79         fail("Should have thrown an exception due XSLT file not found");
80     } catch (Exception e) {
81         // expected
82     }
83 }
```

Listing 1.1: Test code snippet with a ECT test smell.

```
74 @Test
75 public void testErrorListener() throws Exception {
76     RouteBuilder builder = createRouteBuilder();
77     CamelContext context = new DefaultCamelContext();
78     context.getRegistry().bind("myListener", listener);
79     context.addRoutes(builder);
80     assertThrows(RuntimeCamelException.class, () -> context.start());
81 }
```

Listing 1.2: Original refactoring using an `assertThrows` to fix the ECT test smell.

test method of the `XsltCustomErrorListenerTest` test class of the `Camel` project[1]. That test class aims to verify whether a listener logs the errors and throws the exceptions for errors in XSLT files. The `testErrorListener` method creates a listener if the XSLT file exists; otherwise, it throws an exception using a `try/catch` structure (lines 73 - 82). It represents an Exception Handling (ECT) test smell, which can hide real problems and hamper debugging. In addition, the bug report using the fail statement is a bad programming practice that masks the stack trace details (KIM; CHEN; YANG, 2021).

Developers should use the features available in the testing frameworks to automatically pass or fail the test method. It means that developers should use the assert methods (e.g., `assertThat`, `assertThrows`, `assertEquals`, `assertNull`, etc) and annotations (e.g., `@Before`, `@After`, `@Ignore`, `@Rule`, etc) implemented by the JUnit framework (PERUMA *et al.*, 2020a). Listing 1.2 presents the refactoring performed by the developers of the `Camel` project, showing awareness of the design problem. As a refactoring strategy to fix the test smell, the developers used the `assertThrows` method provided by JUnit5 with Java 8 lambda expressions to eliminate the need for manually passing or failing the test method (line 80).

[1]Available at: <https://github.com/apache/camel/>

```
74  @Test(expected=RuntimeCamelException.class)
75  public void testErrorListener() throws Exception {
76      RouteBuilder builder = createRouteBuilder();
77      CamelContext context = new DefaultCamelContext();
78      context.getRegistry().bind("myListener", listener);
79      context.addRoutes(builder);
80      context.start();
81  }
```

Listing 1.3: Alternative refactoring using method annotation to fix the ECT test smell.

```
74  @Test
75  public void testErrorListener() throws Exception {
76      RouteBuilder builder = createRouteBuilder();
77      CamelContext context = new DefaultCamelContext();
78      context.getRegistry().bind("myListener", listener);
79      context.addRoutes(builder);
80      Throwable expect = catchThrowable(() -> context.start());
81      assertThat("XSLT file not found", expect, instanceOf(RuntimeCamelException.class));
82  }
```

Listing 1.4: Alternative refactoring using the `Throwable` class to fix the ECT test smell.

```
74  @Rule
75  public ExpectedException thrown = ExpectedException.none();
76
77  @Test
78  public void testErrorListener() throws Exception {
79      RouteBuilder builder = createRouteBuilder();
80      CamelContext context = new DefaultCamelContext();
81      context.getRegistry().bind("myListener", listener);
82      context.addRoutes(builder);
83      thrown.expect(RuntimeCamelException.class);
84      thrown.expectMessage("XSLT file not found");
85      context.start();
86  }
```

Listing 1.5: Alternative refactoring using `@Rule` annotation to fix the ECT test smell.

Listing 1.3 presents an alternative for refactoring the same test method using the optional `expected` parameter of the `@Test` annotation to handle exceptions (line 74). Listing 1.4 presents a refactoring of the same test method using the `Throwable` class to catch the exception and the `assertThat` method to verify whether the exception is the one expected (lines 80 and 81). Listing 1.5 uses the `@Rule` annotation to specify (lines 74 and 75) and throws the exception and its message (lines 83 and 84). While all those constructs are valid for handling exceptions with JUnit4, an old construct would correspond to potential test smells when migrating the version of the JUnit framework.

When analyzing pull requests to fix ECT test smells, a researcher made a pull request[2] to the `Hadoop` project to understand developers' perceptions about replacing `@Test(expected = IllegalStateException.class)` JUnit4 with the JUnit5 structure `assertThrows(IllegalStateException.class, () -> {...})`.

> **Researcher:**
> (...) For example, the smell in TestSSLFactory.java occurs when exception handling can alternatively be implemented using assertion rather than annotation: using assertThrows(IllegalStateException.class, () -> {...}); instead of @Test(expected = IllegalStateException.class). (...)

Although recognition of using test structures from older versions of the testing framework is suboptimal, the contributor intends to retain the original solution to maintain consistency with the pattern of using AssertJ to handle exceptions. Additionally, the contributor suggests addressing issues such as lacking error messages, reversed arguments, and the use of `try/catch` constructs in tests. The developer even highlights a pull request that lacks explanatory messages[3], underscoring the challenges of manual test refactoring. In the role of a code reviewer, the contributor dedicates significant time to guiding others in meeting test-writing expectations.

> **Contributor:**
> I do think test age is an issue, and old code is suboptimal, but using test(expected) over our own intercept() or even assertThrows isn't something we'd do today. (...)
> if anyone was to go near old tests, i'd target things i really don't like
> 1. assertTrue/assertFalse without error messages or detail why the test failed. Fix: use assertJ assertions with .describedAs(), or at least add messages
> 2. assertEquals with the arguments reversed. fix, swap, or better: assertj.
> 3. code which uses try/catch/fail rather than intercept()

> **Contributor:**
> now, one thing to consider there is: what stylecheckers etc can we use to stop new prs coming in which don't do all of this, or lose stack traces when validating caught exceptions? As all to often, the work of getting a PR in is the time spent teaching people how to write tests that meet my expectations (for me) and the time spent waiting for review, making the changes and repeating (for them). see #6003 as an example. if we have the CI tooling automatically imposing policies on tests, then everyone's time is better used.

[2] <https://github.com/apache/hadoop/pull/5982>
[3] see <https://github.com/apache/hadoop/pull/6003>

The developer's insights underscore the challenge of deciding which refactorings to apply and how to address test smells, indicating a potential need for assistance from an automated test refactoring recommendation tool with a developer-centric approach. This aligns with our approach to analyze supervised ML algorithms to classify developers' intentions in applying test refactoring and the specific test refactoring operation. Currently, our understanding is based on the work of Aniche *et al.* (2022), in which the authors investigated the effectiveness of supervised ML algorithms in predicting refactorings in production code using structural metrics and code smells. While these algorithms demonstrated proficiency in predicting refactoring opportunities in production code, it remains uncertain whether they would perform similarly in identifying refactoring opportunities in test code.

Given the unique characteristics of production and test code, applying the findings of (ANICHE *et al.*, 2022) directly to test code encounters two primary challenges. Firstly, the smells and structural metrics utilized as predictors of refactorings in production code may not directly translate to test code. This disparity complicates the transferability of predictive models from production to test code. Secondly, Fowler's catalog (FOWLER, 1999) does not encompass certain test-specific refactorings proposed by (DEURSEN *et al.*, 2001) and others. Consequently, ML algorithms trained on Fowler's catalog cannot predict these additional refactorings specific to test code.

1.2 RESEARCH STATEMENT

This thesis investigates the feasibility of identifying test refactoring opportunities and appropriate fixes using ML techniques. Our investigation is built upon two objectives:

O1. Mining common test smells and their refactorings in practice. Existing test smells and test refactoring catalogs are primarily based on the researchers' intuition (FOWLER, 1999; DEURSEN *et al.*, 2001; MESZAROS, 2007). Our approach provides a catalog of test smells and test refactoring operations to fix them based on actual development practices. We explored the change history of software repositories to extract the refactoring operations performed on the test code. Analyzing test refactoring operations helps reveal (a) the actual issues developers deemed as test smells and (b) how such issues were fixed (i.e., refactored) in practice.

O2. Identifying refactoring opportunities in test code and proper fixings for test smells. We derived a dataset containing smelly and non-smelly refactored

test codes while we mined commonly used test code refactoring in practice. Then, we used ML techniques to learn from the features extracted from the change history of the test code stored in that dataset.

In order to address the outlined objectives, we have formulated three high-level Research Questions (RQ) to guide this thesis:

RQ$_1$. *How do developers perform test code refactorings to fix test smells in open-source projects?*

In **RQ$_1$**, our objective was to catalog test code refactorings commonly performed by developers in practice to address test smells. To achieve this, we conducted a qualitative analysis on a sample of open-source projects, manually classifying test code changes. Additionally, we investigated developers' preferences regarding refactoring operations for fixing test smells by submitting pull requests to open-source projects.

RQ$_2$. *How do test refactoring operations affect test code quality?*

In **RQ$_2$**, our objective was twofold: i) To determine whether low-quality test classes, as measured by structural metrics and the presence of test smells, are more likely to have test refactorings; ii) To assess the impact of test code refactorings on test code quality. To achieve these goals, we conducted an empirical study involving the analysis of refactorings carried out by developers, mined from the version history of open-source projects.

RQ$_3$. *How accurately can we suggest test refactoring operations for fixing test smells using ML techniques?*

In **RQ$_3$**, our objective was to explore the performance of supervised machine learning (ML) algorithms in classifying developers' intentions regarding test refactoring. This investigation encompassed two distinct classification tasks: i) Classifying code changes where developers would conduct test refactoring; ii) Classifying the specific test refactoring operations applied by developers.

1.3 RESEARCH METHOD

In this research, we employed a combination of various methods to address the research problem and strengthen the conclusions of our study (HESSE-BIBER, 2010). Figure 1.1 outlines four specific components of our research design, detailed as follows:

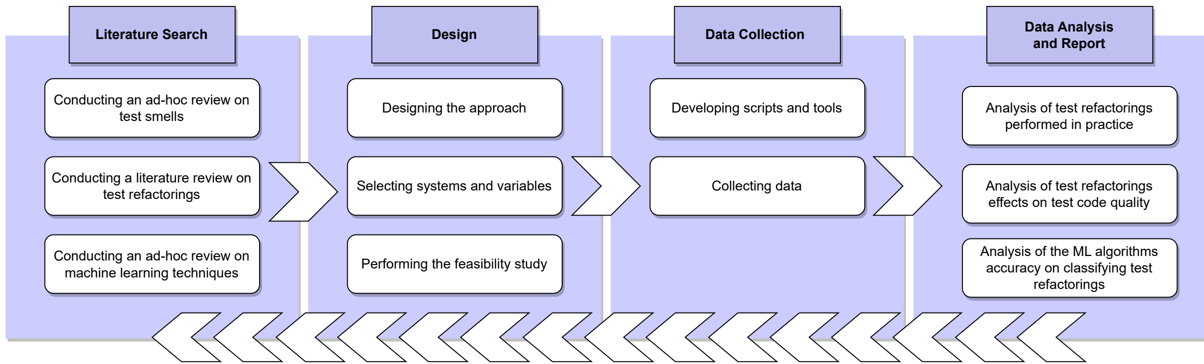


Figure 1.1: Research Method.

- **Literature search.** This phase entails the analysis of the literature on test smells, refactorings, and ML techniques to comprehend the state-of-the-art and establish a foundation for defining our research problem. We conducted ad-hoc searches in the literature and reviews to outline the background concepts (Chapter 2), grasp the current state-of-the-art, and identify any gaps in the literature (Chapter 3);

- **Design.** This phase encompasses experimental design aimed at addressing our research problem. We selected subject systems, identified variables, and conducted feasibility studies for each research question. Figure 1.2 illustrates our approach to investigating these questions through three steps:

  - **(1) Selecting subject projects:** This step consists of selecting software projects from GitHub. We have chosen diverse open-source Java projects with tests written using the JUnit testing framework. *Set #1* comprises 13 open-source Java projects from the Apache Foundation, and we used it to perform a manual analysis of test code. *Set #2* comprises 63 projects from GitHub. It was used for repository mining to gather data on test smells, refactorings, and structural metrics using automated tools. *Set #3* comprises ten projects from GitHub, with information on test smells, refactorings, structural and process metrics. It was used to train and evaluate machine learning models.
  - **(2) Deriving test refactorings:** This step consists of extracting test code changes from *Set #1* of projects (step a) and manually classifying them into test smells and test refactoring operations (step b), resulting in *Dataset #1*. The outcome of this process is a catalog describing the reengineering process required to perform test refactoring (step c). This catalog proved instrumental in developing rules to enhance existing repository mining tools for detecting

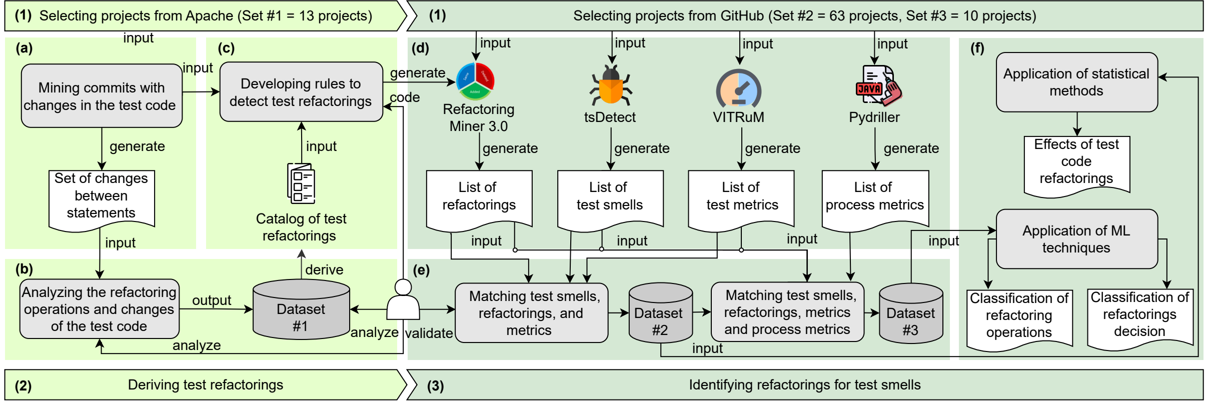


Figure 1.2: Overview of our approach.

test refactorings (see Chapter 5).

– **(3) Identifying refactoring operations for test smells:** This step involves collecting data using automated tools (step d). In more detail, we collected information on test smells, test refactoring operations, and structural metrics from *Set #2*, generating *Dataset #2* (step e). In addition, it involves the collection of process metrics from *Set #3* to generate *Dataset #3* (step e). Finally, we analyzed the data from both datasets (step f). We utilized *Dataset #2* to analyze the impacts of test code refactorings on test code quality (refer to Chapter 5), and *Dataset #3* to identify refactoring opportunities and specific operations developers would undertake in the test code (refer to Chapter 6). It is worth noting that each dataset was used to investigate a specific research question and, therefore, offers complementary insights for our analysis.

- **Data Collection:** This phase involves developing the requisite tools, scripts, and models to conduct the experiments. We automated the procedures employed in the feasibility study to generate a comprehensive dataset comprising pairs of smelly and refactored test codes. Specifically, we expanded existing automated tools for repository mining to gather data on test smells, test-specific refactorings, and structural and process metrics (refer to Chapter 5 and Chapter 6).

- **Data Analysis and Reporting:** This phase involves analyzing the data collected to address our research questions. Firstly, we analyzed the curated dataset of test code refactorings carried out in practical scenarios to catalog the refactorings (refer to Chapter 4). Secondly, we examined the effects of test code refactorings on code

quality using the extensive dataset mined from the version history of open-source projects (refer to Chapter 5). Finally, we employed the large dataset as input for ML techniques to extract features and classify whether developers would undertake refactoring operations and determine which specific test-specific refactorings they would apply (refer to Chapter 6).

1.4 OVERVIEW OF THE PROPOSED SOLUTION BY ARIES LAB

This thesis is situated within the context of the ARIES Lab, which strives to develop a comprehensive framework for detecting test smells and refactoring test codes. This framework takes into account developers' perceptions and daily practices that influence the introduction and elimination of test smells. Figure 1.3 illustrates an abstract layered flow synthesized from the analysis of our past, present, and potential future work. It has six main approaches:

- **Rules-based test smell detection (Figure 1.3 - steps 1.1 to 1.3):** This approach utilizes the definition of test smells to derive rules for their detection. It begins by generating the test code model through an Abstract Syntax Tree (AST) (step 1.1). Subsequently, it employs conditional expressions to formulate the detection rules (step 1.2). Consequently, the approach detects a set of test smells when the defined rules are satisfied (step 1.3).
- **Metrics-based test smell detection (Figure 1.3 - steps 2.1 to 2.3):** This approach employs metrics to characterize test smells. It starts by generating the source code model through an AST (step 2.1), then identifies a set of metrics and applies suitable thresholds (step 2.2). Consequently, the approach detects a set of test smells when the metric values satisfy the defined thresholds (step 2.3).
- **Machine Learning-based test smell detection (Figure 1.3 - steps 3.1 to 3.6):** This approach utilizes the source code model (step 3.1) and existing examples (step 3.2) to detect a set of metrics characterizing test smells in the test code (step 3.3). Subsequently, it instantiates an ML model (step 3.4) and employs a set of ML algorithms (step 3.5) to learn from the test code metrics. Consequently, it classifies the test smells in the test code based on the test code metrics (step 3.6).
- **Rules-based refactoring recommendation (Figure 1.3 - steps 4.1 to 4.8):** Similar to the Rules-based test smell detection approach, this method utilizes the

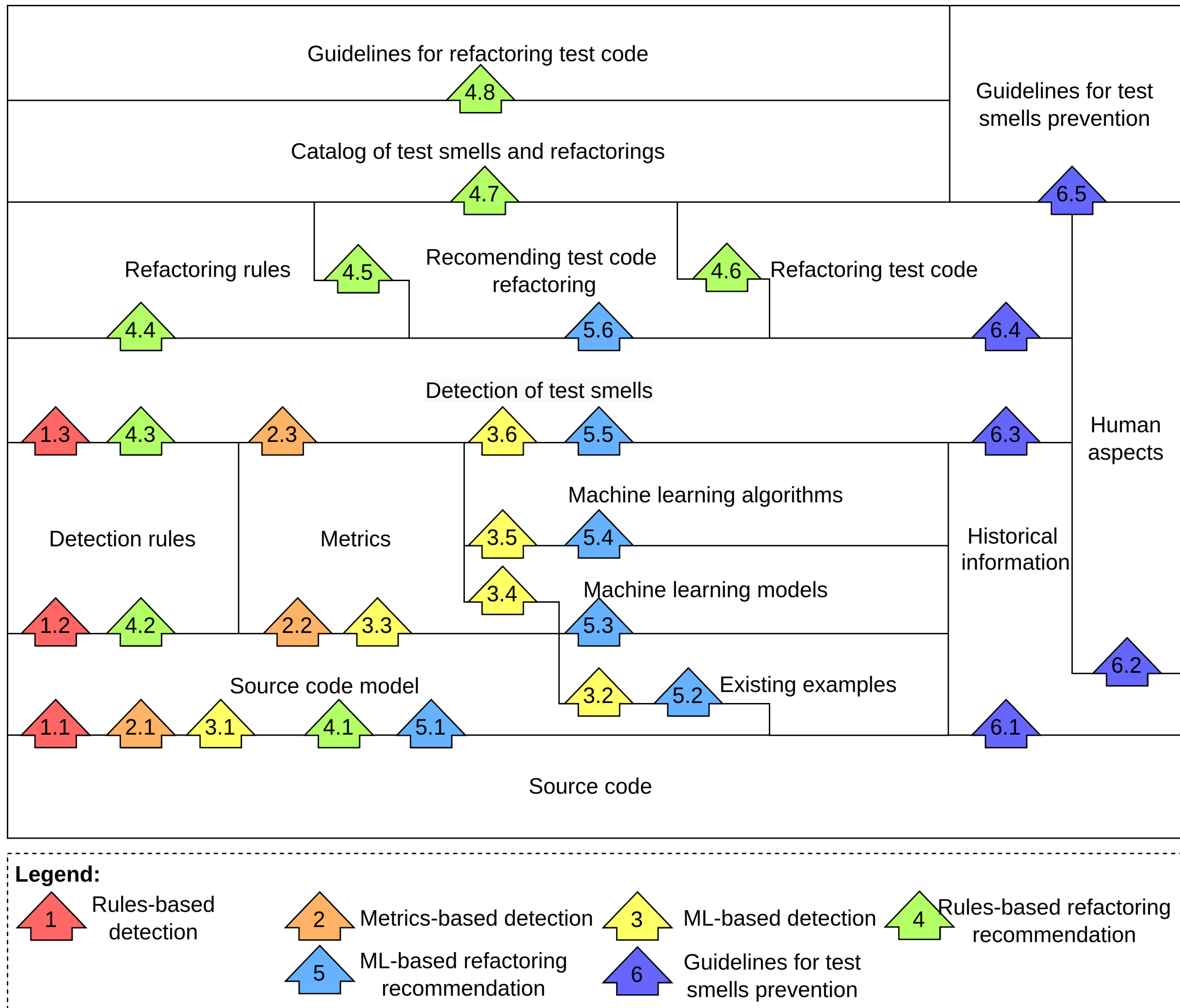


Figure 1.3: Overview of the ARIES Lab framework.

definition of test smells to apply a set of rules in the test code model for detecting test smells (steps 4.1, 4.2, and 4.3). It then matches the detected test smells with their refactoring rules (step 4.4) and recommends the refactorings to developers (step 4.5). Test code refactoring occurs once developers accept the recommendation (step 4.6). Understanding the detection and refactoring of test smells can help derive a catalog and guidelines for refactoring test smells (steps 4.7 and 4.8).

- **Machine Learning-based refactoring recommendation (Figure 1.3 - steps 5.1 to 5.6):** This approach utilizes the test code model (step 5.1) and existing examples (step 5.2) containing pairs of smelly and refactored test codes to instantiate an ML model (step 5.3). Next, it applies ML algorithms (step 5.4) in the model to detect the test smells (step 5.5). Consequently, it recommends proper refactorings

to fix test smells in the test code (step 5.6).

- **Guidelines for test smells prevention (Figure 1.3 - steps 6.1 to 6.5):** This approach utilizes historical information from the test code (step 6.1) to extract information about human aspects (step 6.2), such as the frequency of developers' contributions and experience in open-source projects, and the insertion and removal of test smells from the test code (steps 6.3 and 6.4). Consequently, it aims to derive guidelines for preventing test smells (step 6.5).

Scope Definition. We limited the scope of this thesis to the *Machine Learning-based refactoring recommendation* (steps 5.1 to 5.6). We investigate the ML techniques considering occurrences of test smells in refactored and non-refactored test code to discover the developers' intention to apply test refactoring. Rather than refactoring the test code, we expect a ML-based solution for classifying which refactoring operations developers would perform in the test code, which can later blend with the *Rules-based refactoring recommendation* approach. It is worth highlighting that our research has yielded significant contributions regarding rules-based refactoring recommendations, such as tools for mining repositories and a catalog of refactorings for test smells (steps 4.1 - 4.5, 4.7).

1.5 THESIS OUTLINE

Figure 1.4 shows a schematic overview of the thesis structure. Besides the Introduction chapter, we outlined the remaining five chapters as follows.

- **Background and Related work.** This part presents the concepts of test smells, refactorings, and ML techniques. In addition, it presents studies focusing on related problems.
  - **Background (Chapter 2)** provides the background concepts for the foundation of knowledge on the topics involved in this investigation, namely test smells, refactoring, and ML.
  - **Related work (Chapter 3)** presents the current state-of-the-art research concerning test smells, reveals limitations in present tools and techniques, and identifies research opportunities.
- **Understanding test refactorings.** This part is divided into three main studies

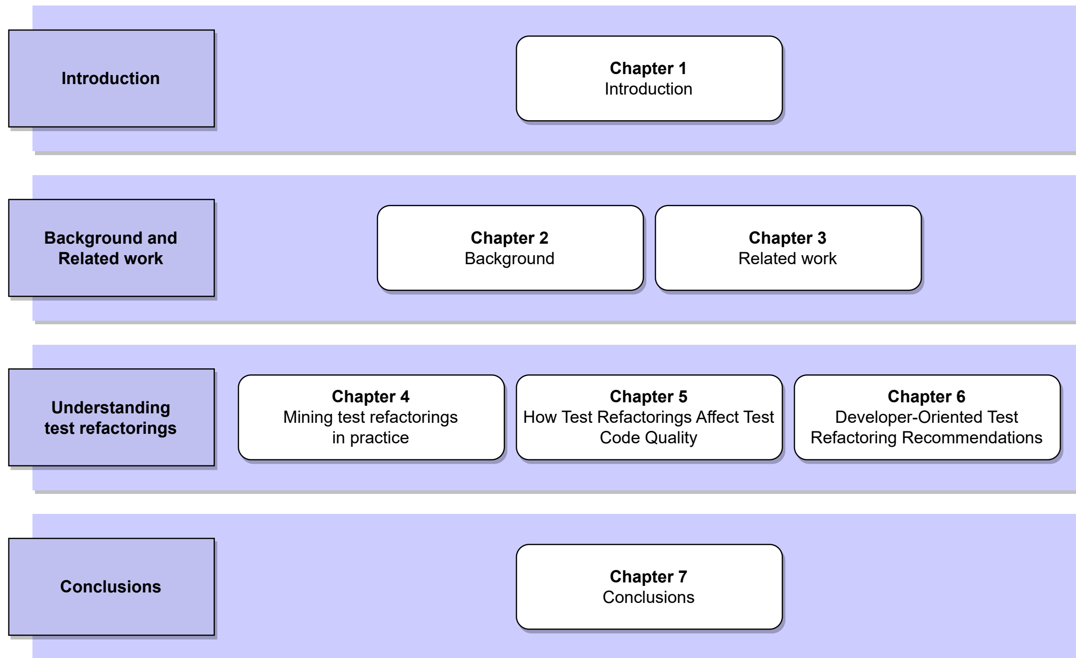


Figure 1.4: Schematic overview of the thesis structure.

connected to the RQs, aiming to provide an understanding of the test refactorings fix test smells.

- **Mining test refactorings in practice (Chapter 4)** presents an exploratory empirical study on open-source projects from GitHub to analyze test refactorings. It reveals i) the actual issues developers deemed as test smells and ii) how they were fixed (i.e., refactored) in practice.
- **How Test Refactorings Affect Test Code Quality (Chapter 5)** presents an empirical study to analyze low-quality test classes regarding structural metrics and test smells. In addition, it provides indications on which test classes are more likely to be refactored and to what extent test refactoring operations are effective in improving code quality.
- **Developer-Oriented Test Refactoring Recommendations (Chapter 6)** presents an empirical study to investigate the performance of supervised ML algorithms in classifying the developers' intention to apply test refactoring and which developers apply test-refactoring operations.

- **Conclusions (Chapter 7).** This part summarizes the achieved contributions and discusses the perspectives on future research directions.

Chapter 2

BACKGROUND

Manual or automated software testing aims to demonstrate that the software works as expected, i.e., meeting customer needs and finding errors (QUADRI; FAROOQ, 2010; AMANNEJAD *et al.*, 2014). Manual software testing requires a tester to manually interact with the software under test to verify its behavior. Automated software testing requires the development of test scripts to verify the behavior of the software under test. Although both approaches detect defects in the software, automated software testing has some benefits compared with manual software testing, such as repeatability, predictability, and efficiency in test runs (GAROUSI; AMANNEJAD; BETIN CAN, 2015). Therefore, the software industry predominantly uses automated testing as a safe way to maintain software, i.e., fixing bugs, adding new features, and improving code quality (BOWES *et al.*, 2017).

Developing test scripts for automated software testing is a nontrivial task. Developers can use testing tools and frameworks (e.g., JUnit[1]) to facilitate the development of test scripts and the interpretation of their results (GAROUSI; AMANNEJAD; BETIN CAN, 2015; BOWES *et al.*, 2017). Even though developers can adopt bad design practices to either design or implement the test code, leading to the insertion of test smells (BAVOTA *et al.*, 2015). Test smells can decrease the test code quality, harming the testing and maintenance activities (BAVOTA *et al.*, 2015; PALOMBA *et al.*, 2016; SPADINI *et al.*, 2020). Therefore, it is essential to employ practices and tools that could support fixing test smells properly.

[1] <https://junit.org/>

This chapter introduces the fundamentals of automated software testing, test smells, test refactorings, and techniques for handling test smells. Section 2.1 briefly overviews automated software testing, test frameworks, and test code. Section 2.2 presents the definitions and examples of test smells. Section 2.3 presents the definition and types of refactoring recommendation tools to support developers handling test smells. Section 2.4 presents structural and process metrics to evaluate the test code quality. Section 2.5 presents the Machine Learning (ML) techniques.

2.1 AN OVERVIEW OF AUTOMATED SOFTWARE TESTING

Amannejad *et al.* (2014) categorized software testing into four key activities:

- **Test case design.** It involves designing test cases to satisfy coverage criteria or other engineering goals, involving sub-activities such as identifying test data, including test inputs and expected test outputs;
- **Test scripting.** It involves documenting test cases in manual software testing or developing test code in automated software testing. Many test tools and frameworks support test scripting (GAMIDO; GAMIDO, 2019), e.g., IBM Rational Manual Tester[2] for manual software testing and JUnit for automated software testing;
- **Test execution.** It consists of running test cases (a set of tests performed on the same unit software under test) on the software under test and recording the outputs. There is a dependency between the test scripting and test execution regarding using a manual or automated approach, i.e., the test execution always uses the same approach as the test scripting;
- **Test evaluation.** It consists of evaluating the testing results (pass or fail). A human tester can analyze the results against the expected ones or incorporate verification points in the test code.

We can perform those four activities manually or automatically. However, manually developing tests is time and effort-consuming and can only sometimes be as effective as automated tests in finding defects (GAROUSI; AMANNEJAD; BETIN CAN, 2015).

[2] <https://www.ibm.com/support/pages/ibm-rational-manual-tester-version-7017>

Automated software testing is more effective, allowing test execution quickly and repeatedly. In this direction, the software industry predominately uses it as a cost-effective way for the regression testing of software with a long maintenance life (GAROUSI; AMANNEJAD; BETIN CAN, 2015; BOWES *et al.*, 2017).

Automated software testing requires testing tools or frameworks to test the software under test. A testing framework is a software library that helps to standardize the test specification, execution, and reporting (ROMPAEY *et al.*, 2007). In particular, a test script is a code written in programming languages using a testing framework (e.g., JAVA and JUNIT) executed on the software to analyze its result against the expected output (GAROUSI; AMANNEJAD; BETIN CAN, 2015). A test script usually codes a given test case into a consistent structure that involves the following (SSVT) steps executed in sequence (ROMPAEY *et al.*, 2007):

- **Setup (S).** The test fixture sets up the production code in the desired state. It instantiates the production code using attributes, setup methods, and test data;
- **Stimulate (S).** The test code sends a stimulus for the production code to provoke an action (a state change), i.e., the stimulus refers to any object manipulation in the test code that is not an assertion;
- **Verify (V).** The test code queries the production code and fetches the result of the stimulus through assertions. The assertions report the test outcome and do not make any change in the production code;
- **Teardown (T).** All the resources used to instantiate, stimulate, and verify the production code are released to avoid dependencies among different tests.

Listing 2.1 presents all the above steps in the `EndpointTest` test class from the `Camel` project[3], written with the JUNIT framework. The test class uses the Java Management Extensions (JMX) to set up the context (lines 30, 32-37 - Setup step), uses the context to get the status of a Beanstalk job (line 41 - Stimulate step), and verifies whether the status is not null and its priority (lines 42 and 43 - Verify step), then puts the context back into the initial state (lines 77-80 - Teardown step).

[3]Available at: <https://github.com/apache/camel/>

```
29 public class EndpointTest {
30     CamelContext context;
31
32     @BeforeEach
33     public void setUp() throws Exception {
34         context = new DefaultCamelContext(false);
35         context.disableJMX();
36         context.start();
37     }
38
39     @Test
40     void testPriority() {
41         BeanstalkEndpoint endpoint = context.getEndpoint("beanstalk:default",
               BeanstalkEndpoint.class);
42         assertNotNull(endpoint, "Beanstalk endpoint");
43         assertEquals(1000, endpoint.getJobPriority(), "Priority");
44     }
...
77     @AfterEach
78     public void tearDown() {
79         context.stop();
80     }
81 }
```

Listing 2.1: An example of setup, stimulation, verify, and teardown in test code.

2.2 TEST SMELLS

Test smells have gained importance over the last few years among researchers and practitioners. Deursen *et al.* (2001) earlier introduced the concept of test smells to denote poorly designed test codes. Meszaros, Smith and Andrea (2003) broaden that concept, specifying seven test smells related to the test code level and five test smells related to the test behavior. Later, several researchers extended the catalog of test smells (GREILER *et al.*, 2013; PALOMBA *et al.*, 2016), investigated the impacts of test smells on the test code quality (BAVOTA *et al.*, 2015; TUFANO *et al.*, 2016), and proposed solutions to handle test smells (VIRGINIO *et al.*, 2020; PERUMA *et al.*, 2020a). The remaining sections present examples of test smells, strategies, and automated tools to handle them.

2.2.1 Test smells definition and examples

Garousi and Küçük (2018) conducted a multivocal literature mapping on formal and informal sources. They obtained 196 test smells, mainly discussed among practitioners in informal sources. The authors classified them into eight high-level categories: i) Test execution/behavior, ii) Test semantic/logic, iii) Design related, iv) Issues in test steps, v) Mock and stub related, vi) In association with production code, vii) Code related, and viii) Dependencies. While the detection of certain test smells occurs at the method level

and does not require code execution, other test smells require viewing the entire test class (e.g., the *Lazy Test*) or even executing the test suite (e.g., the *Slow Test*) (Figure 2.1). In this thesis, we considered 21 test smells detected by tsDetect[4], a tool that presents the highest accuracy among the existing detection tools (ALJEDAANI *et al.*, 2021). Next, we describe the test smells that most contributed to our approach, along with examples within the context of the `Camel` project.

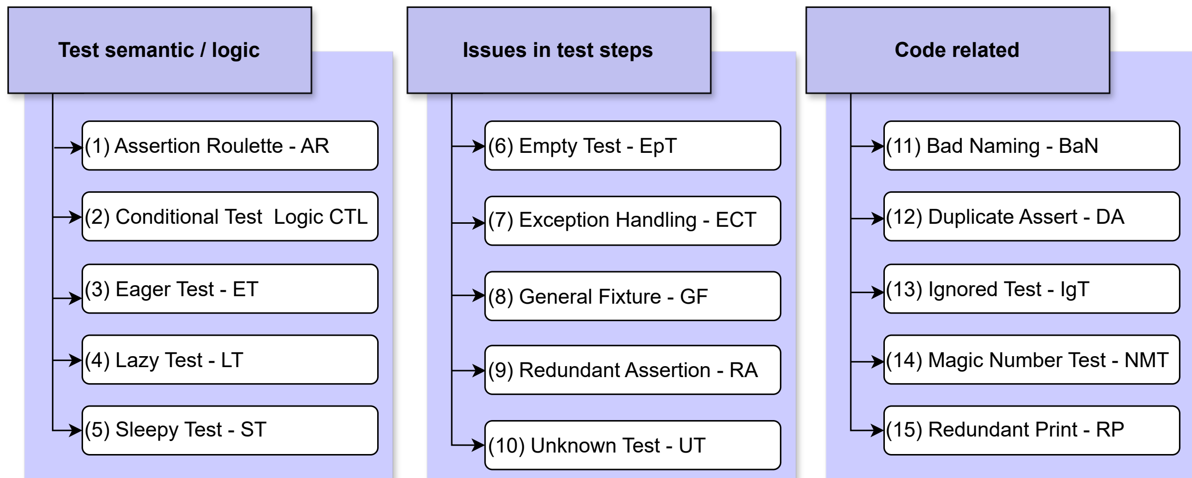


Figure 2.1: Garousi and Küçük (2018)'s analyzed test smells at test method level.

Test semantic/logic

This category refers to test smells related to test logic and several responsibilities per test (GAROUSI; KüçüK, 2018; MARTINS *et al.*, 2023). Some test smells in this category are described as follows.

(1) Assertion Roulette (AR). It occurs when a test method has multiple assertions without an explanation message, making it hard to understand the goal of the assertion (DEURSEN *et al.*, 2001). Listing 2.2 presents an AR test smell in the `testNoMatch1ThenMatchingJustException` method. That test method verifies whether an `ExceptionPolicy` object equals another through an `assertEquals` assertion (line 162). Still, there is no string message (1st or 3rd parameter according to particularities of the JUnit versions) in the assertion explaining its goal.

[4]Available at: https://testsmells.org/pages/testsmelldetector-architecture.html

```
48  public class DefaultExceptionPolicyStrategyTest {
49      private ExceptionPolicy type2;
50
...
158     @Test
159     public void testNoMatch1ThenMatchingJustException() {
160         setupPolicies();
161         ExceptionPolicy result = findPolicy(new AlreadyStoppedException());
162         assertEquals(type2, result);
163     }
164 }
```

Listing 2.2: An example of the AR test smell.

```
39  @Test
40  public void testRoute() {
41      Route route = context.getRoutes().get(0);
42      DefaultRoute consumerRoute = assertIsInstanceOf(DefaultRoute.class, route);
43
44      Processor processor = unwrap(consumerRoute.getProcessor());
45      Pipeline pipeline = assertIsInstanceOf(Pipeline.class, processor);
...
49      for (Processor child : pipeline.next()) {
50          Channel channel = assertIsInstanceOf(Channel.class, child);
51          assertNotNull(channel.getErrorHandler(), "There should be an error handler");
52          assertIsInstanceOf(DefaultErrorHandler.class, channel.getErrorHandler());
53      }
54  }
```

Listing 2.3: An example of the CTL test smell.

(2) Conditional Test Logic (CTL). It occurs when a test method contains one or more control statements, whereas it should be simple and execute all statements. Tests with branch points require greater care when analyzing whether the test is correct (MESZAROS, 2007). Listing 2.3 presents a CTL test smell in the `testRoute` test method of the `DefaultErrorHandlerTest` test class. It verifies whether there is a default error handler in the pipeline using the `assertNotNull` test method inside a loop (lines 49-53).

(3) Eager Test (ET). It occurs when a test method invokes several methods of the production code, making the test code hard to understand and more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain (DEURSEN *et al.*, 2001). Listing 2.4 shows the ET test smell in the `testBytesSourceCtr` test method of the `BytesSourceTest` test class. That test method verifies different attributes of the `bs` object, calling the `getData()`, `getSystemId()`, `getInputStream()`, and `getReader()` production methods (lines 30, 32, 34, and 35).

```
27  @Test
28  public void testBytesSourceCtr() {
29      BytesSource bs = new BytesSource("foo".getBytes());
30      assertNotNull(bs.getData());
31      assertEquals("BytesSource[foo]", bs.toString());
32      assertNull(bs.getSystemId());
33
34      assertNotNull(bs.getInputStream());
35      assertNotNull(bs.getReader());
36  }
```

Listing 2.4: An example of the ET test smell.

```
231 @Test
232 public void testParseLoadBalance() throws Exception {
233     RouteDefinition route = assertOneRoute("routeWithLoadBalance.xml");
234     assertFrom(route, "seda:a");
235     LoadBalanceDefinition loadBalance = assertOneProcessorInstanceOf(LoadBalanceDefinition.
            class, route);
236     assertEquals(3, loadBalance.getOutputs().size(), "Here should have 3 output here");
...
239 }
240
241 @Test
242 public void testParseStickyLoadBalance() throws Exception {
243     RouteDefinition route = assertOneRoute("routeWithStickyLoadBalance.xml");
244     assertFrom(route, "seda:a");
245     LoadBalanceDefinition loadBalance = assertOneProcessorInstanceOf(LoadBalanceDefinition.
            class, route);
246     assertEquals(3, loadBalance.getOutputs().size(), "Here should have 3 output here");
...
252 }
```

Listing 2.5: An example of the LT test smell.

(4) Lazy Test (LT). It occurs when multiple test methods invoke the same production method, making the test code hard to maintain as responsibilities related to one production method are tested across different test methods (DEURSEN *et al.*, 2001). Listing 2.5 shows the LT test smell in the `testParseLoadBalance` and `testParseStickyLoadBalance` test methods of the `XmlParseTest` test class. Those test methods call the same production methods (lines 233 - 236 and 243 - 246).

(5) Sleepy Test (ST). It occurs when a test method explicitly calls a thread sleep. Different devices might have different processing times for a task, leading to unexpected results (PERUMA *et al.*, 2019). Listing 2.6 presents the ST test smell on line 62 occurring of the `testGrouped` test method of the `AggregateGroupedExchangeBatchSizeTest` class. In that case, after sending the initial set of messages, the test waits for one second before checking for additional received messages.

```
37 @Test
38 public void testGrouped() throws Exception {
...
58     assertEquals("100", grouped.get(0).getIn().getBody(String.class));
59     assertEquals("150", grouped.get(1).getIn().getBody(String.class));
60
61     // wait a bit for the remainder to come in
62     Thread.sleep(1000);
63
64     if (result.getReceivedCounter() == 2) {
...
72         assertEquals("130", grouped.get(0).getIn().getBody(String.class));
73         assertEquals("200", grouped.get(1).getIn().getBody(String.class));
74     }
75 }
```

Listing 2.6: An example of the ST test smell.

```
59 @Test
60 public void testSendAndReceive() throws Exception {
61 }
```

Listing 2.7: An example of the EpT test smell.

Issues in the test steps

This category refers to test smells occurring in specific language constructs, such as assertions and setup methods, leading to incomplete test steps. (GAROUSI; KüçüK, 2018; MARTINS *et al.*, 2023). Some test smells in this category are described as follows.

(6) Empty Test (EpT). It occurs when a test method has no executable statements. An empty test can be considered more dangerous than not having a test case since JUnit will indicate that the test passes even if there are no executable statements in the method body (PERUMA *et al.*, 2019). Listing 2.7 presents an EpT test smell in the `testSendAndReceive` test method with no executable statements (lines 60 and 61).

```
85 @Test
86 public void testMandatoryConvertToNotPossible() {
87     try {
88         CamelContextHelper.mandatoryConvertTo(context, CamelContext.class, "5");
89         fail("Should have thrown an exception");
90     } catch (IllegalArgumentException e) {
91         // expected
92     }
93 }
```

Listing 2.8: An example of the ECT test smell.

```
34 @BeforeEach
35 public void setUp() {
36     camelContext = new DefaultCamelContext();
37     Message message = new DefaultMessage(camelContext);         message.setBody("This is the
           message body");
38     exchange = new DefaultExchange(camelContext);
39     exchange.setIn(message);
40     exchangeFormatter = new DefaultExchangeFormatter();
41 }
42
43 @Test
44 public void testDefaultFormat() {
45     String formattedExchange = exchangeFormatter.format(exchange);
46     assertTrue(formattedExchange.contains("This is the message body"));
47 }
```

Listing 2.9: An example of the GF test smell.

```
73 @Test
74 public void testLine1LF() throws Exception {
75     assertReadAsWritten("line1LF", "line1\n", "line1\n");
76 }
```

Listing 2.10: An example of the RA test smell.

(7) Exception Handling (ECT). It occurs when a test method for which pass or failure depends on the production/test method to throw an exception instead of using the testing framework constructs (PERUMA *et al.*, 2019). Listing 2.8 shows the `testMandatoryConvertToNotPossible` test method of the `CamelContextHelperTest` class. It contains an ECT test smell as it uses a `try/catch` block (lines 87-92) to capture the exception thrown by the `mandatoryConvertTo()` production method (line 88).

(8) General Fixture (GF). It occurs when not all test methods use the test case fixture, indicating that the setup is too general. Consequently, it may slow down the test execution as the setup may become complex, with unnecessary steps for tests that don't require them (PERUMA *et al.*, 2019). Listing 2.9 shows the setup of the test class (`DefaultExchangeFormatterTest`) instantiating objects, but the `testDefaultFormat` test method does not use them, e.g., the `exchangeFormatter` object (line 40).

(9) Redundant Assertion (RA). It occurs when a test method contains assertions whose results are always true or are always false. A test must return a binary outcome of whether the intended result is or is not correct and should not return the same output regardless of the input (PERUMA *et al.*, 2019). Listing 2.10 presents an RA test smell in the `testLine1LF` test method of the `IOHelperTest` test class. The assertion contains

```
64 @Test
65 public void testAddDuplicateTypeConverter() {
66     DefaultCamelContext context = new DefaultCamelContext();
67 
68     context.getTypeConverterRegistry().addTypeConverter(MyOrder.class, String.class, new
           MyOrderTypeConverter());
69     context.getTypeConverterRegistry().addTypeConverter(MyOrder.class, String.class, new
           MyOrderTypeConverter());
70 }
```

Listing 2.11: An example of the UT test smell.

```
23 public class OnCompletionAfterChainedSedaRoutes extends ContextTestSupport {
24 
25     @Test
26     public void testOnCompletionChained() throws Exception {
...
84 }
```

Listing 2.12: An example of the BaN test smell.

three parameters: the assertion message, the expected value, and the actual result (line 75). However, the expected and actual values are the same strings.

(10) Unknown Test (UT). It occurs when a test method does not contain assertions. As a result, JUnit shows the test method as passing unless the statements raise an exception in the test method (PERUMA *et al.*, 2019). Listing 2.11 presents a UT test smell in the `testAddDuplicateTypeConverter` test method of the `TypeConverterRegistryTest` test class. That method contains executable statements but not assertions (lines 65-69).

Code Related

This category refers to test smells related to test code duplication, long, complex, and hard-to-understand tests, and tests that do not follow coding best practices regarding naming conventions and code organization (GAROUSI; KüçüK, 2018; MARTINS *et al.*, 2023). Some test smells in this category are described as follows.

(11) Bad Naming (BaN). It occurs when the test classes do not follow the JUnit naming conventions (MESZAROS; SMITH; ANDREA, 2003), i.e., the test class should be in the same package hierarchy as its respective production class, and the test class name is the production class name append or pre-pend with the "Test" word. Listing 2.12 shows the test class `OnCompletionAfter ChainedSedaRoutes` extending `ContextTestSupport`; however, its name does not contain the "Test" word (line 23).

```
851 @Test
852 public void testReset() throws Exception {
853     NotifyBuilder notify = new NotifyBuilder(context).whenExactlyDone(1).create();
854 
855     template.sendBody("direct:foo", "Hello World");
856     assertEquals(true, notify.matches());
857 
858     template.sendBody("direct:foo", "Bye World");
859     assertEquals(false, notify.matches());
860 
861     // reset
862     notify.reset();
863     assertEquals(false, notify.matches());
864 
865     template.sendBody("direct:foo", "Hello World");
866     assertEquals(true, notify.matches());
867 
868     template.sendBody("direct:foo", "Bye World");
869     assertEquals(false, notify.matches());
870 }
```

Listing 2.13: An example of the DA test smell.

```
95  @Disabled("Manually enable this once you configure the parameters in the placeholders above")
96  @Test
97  public void testListUsers() throws Exception {
...
111 }
```

Listing 2.14: An example of the IgT test smell.

(12) Duplicate Assert (DA). It occurs when a test method that checks the same condition with different values. Whether the test method needs to test the same condition using different values,the developers should create a new test method (PERUMA *et al.*, 2019). Listing 2.13 presents a DA test smells in the `testReset` test method of the `NotifyBuilderTest`. That method instantiates a `NotifyBuilder` object and verifies whether the message sent matches the notification (lines 855-859). The same test method calls the `reset` statement (line 861). It repeats the same assertions to verify whether the message sent matches the notification (lines 865-869). In addition, such assertions also contain the AR test smell.

(13) Ignored Test (IgT). It occurs when developers suppress test methods from running using the tags `@Ignore` or `@Disabled`. Consequently, the ignored test methods add unnecessary overhead regarding compilation time, making it hard to understand the test code (PERUMA *et al.*, 2019). Listing 2.14 shows the `testListUsers` method of the `ListUsersFunctionalTest` test class with the @Disabled tag, with a comment to enable the test after configuring the parameters (line 95).

```
57 @Test
58 public void testTimeout1() throws Exception {
59     initResequencer(500, 10);
60     resequencer.insert(4);
61     assertNull(buffer.poll(250));
62     assertEquals((Object) (Integer) 4, (Object) buffer.take());
63     assertEquals((Object) (Integer) 4, (Object) resequencer.getLastDelivered());
64 }
```

Listing 2.15: An example of the MNT test smell.

```
33 @Test
34 public void testGenerateUUID() {
35     ClassicUuidGenerator uuidGenerator = new ClassicUuidGenerator();
36
37     String firstUUID = uuidGenerator.generateUuid();
38     String secondUUID = uuidGenerator.generateUuid();
39     System.out.println(firstUUID);
40
41     assertNotSame(firstUUID, secondUUID);
42 }
```

Listing 2.16: An example of the RP test smell.

(14) Magic Number Test (MNT). It occurs when assert statements in a test method contain numeric literals as parameters that cannot provide their meaning. Magic numbers should be replaced by a named constant, where the name describes where the value comes from or what it represents (PERUMA *et al.*, 2019). Listing 2.15 present MNT test smells in the `testTimeout1` test method of the `ResequencerEngineTest` test class. That method uses numeric literals to verify buffer size and timeout (lines 59-63).

(15) Redundant Print (RP). It occurs when a test method contains print statements. It can consume computing resources or increase execution time if the developer calls a long-running method from within the print method (PERUMA *et al.*, 2019). Listing 2.16 presents a RP test smell in the `testGenerateUUID` test method of the `ClassicUuidGeneratorTest` test class. There is a call for the `System.out.println()` method to print the string value assigned to the `firstUUID` object (line 39). In addition, the test method has an AR test smell (line 41).

2.2.2 Approaches to handle test smells

In their work, Garousi and Küçük (2018) introduced a lifecycle for the manifestation of test smells and outlined approaches to address them. The lifecycle begins with the development of automated software tests, where developers have the option to either utilize

automated tools to design and code the test suite or proceed with manual implementation. When manually coding the test suite, developers should adhere to testing guidelines to prevent the introduction of test smells (MESZAROS; SMITH; ANDREA, 2003). However, adherence to these guidelines is not always consistent, leading to the insertion of test smells. Some test smells are indicative of defects in the test code, such as tests that consistently pass or non-deterministic tests like flaky tests (BELL *et al.*, 2018; FATIMA; GHALEB; BRIAND, 2022). Therefore, developers are encouraged to utilize catalogs and tools for detecting and eliminating test smells to enhance the quality of the test code. As a software project evolves and already includes a test suite, developers must maintain it to test new or modified features. Figure 2.2 outlines the lifecycle, which comprises three key activities:

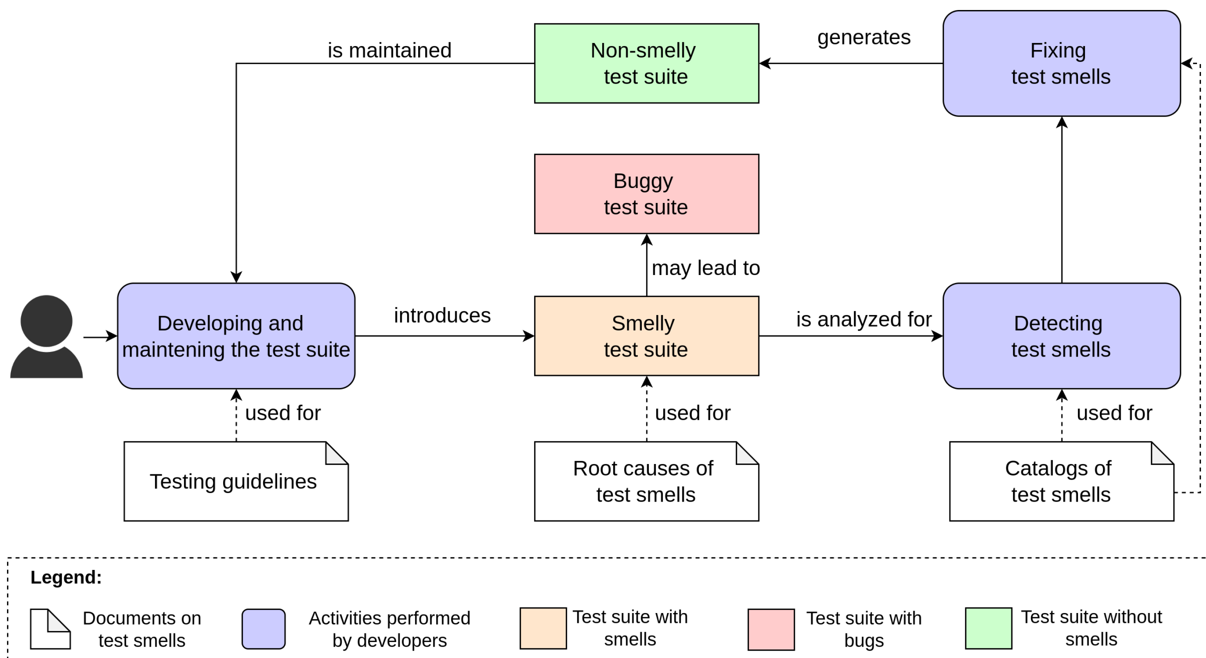


Figure 2.2: Lifecycle of test smells (adapted from Garousi and Küçük (2018)).

(1) Developing and maintaining the test suite. Developers should follow good practices in designing or implementing the test code to prevent the insertion of test smells. The adherence to the Setup, Stimulate, Verify, and Teardown (SSVT) structure is beneficial for the key test design criteria (MESZAROS; SMITH; ANDREA, 2003; MESZAROS, 2007):

- **Concise**. Tests should be standardized in execution behavior, reporting, and error processing;

- **Self Checking**. Tests should report their results without human interpretation;
- **Repeatable**. Tests should run several times without human intervention;
- **Robust**. Tests should run several times and always produce the same result;
- **Sufficient**. Tests should verify all the requirements of the software under test;
- **Necessary**. Tests should not be duplicated;
- **Clear**. Tests should be easy to understand;
- **Efficient**. Tests should run in a suitable amount of time;
- **Specific**. Failing tests should point to a piece of broken functionality to provide defect triangulation;
- **Independent**. Tests should run without depending on other tests in any order;
- **Maintainable**. Tests should follow the design principles of object-oriented;
- **Traceable**. It should be easy to trace the requirement the tests verify.

(2) Detecting test smells. Assessing the test design criteria of the prior item is a non-trivial task. The catalogs of test smells can facilitate the detection of potential violations of those criteria by exploiting their relationship with the SSVT structure (ROMPAEY *et al.*, 2007). While some test smells are generic (independent of any test framework/tool), others are specific to certain frameworks/tools. In this direction, Garousi and Küçük (2018) presented a unified catalog of 196 test smells discussed in the formal and informal literature. Still, the manual detection of test smells can become challenging for large test suites in practice. In addition, several automated tools have been proposed in the literature to support developers in detecting test smells. Besides 22 tools that implement different strategies to detect test smells mapped by Aljedaani *et al.* (2021), other tools are also available (Table 2.1). Usually, the automated tools implement the following detection strategies (ALJEDAANI *et al.*, 2021):

- **Metrics-based.** The test code is parsed and converted into an Abstract Syntax Tree (AST). The AST is analyzed to calculate structural and semantic metrics; then the metrics are interpreted according to threshold values;

- **Rules or Heuristic-based.** The rules or heuristic-based test smell detection approach augments the metrics-based approach by combining the metrics values with patterns found in the test code;

- **Information Retrieval.** The information retrieval-based approach consists of extracting textual content (e.g., source code identifier and source comments) from the test code and normalizing it. The textual content is taken as pre-processed features to serve as input to the ML algorithms;

- **Dynamic Tainting.** The dynamic tainting-based approach monitors the test code while it executes. It runs the test code with a predefined value to detect test smells.

(3) Fixing test smells. Fixing test smells occurs mostly through refactorings. Fowler (1999) presents a catalog of refactoring to improve the production code quality. Later, Deursen *et al.* (2001) extended such a catalog with refactorings for the test code. As the test code refactoring to fix test smells can become challenging for large test suites in practice, other studies have proposed tools to support developers in refactoring the test code (ALJEDAANI *et al.*, 2021). Table 2.1 shows which detection tools support refactoring strategies. The TestHound tool provides textual information for the test smell refactoring. Differently, RAIDE, RTj, DARTS, and TRex tools provide a (semi) automated refactoring to fix test smells. However, those tools neither provide details concerning the accuracy nor support recent features of JUnit5.

2.3 SOFTWARE REFACTORING

Refactoring is one of the most important activities to improve code quality. The objectives of refactoring include but are not limited to (MENS; TOURWE, 2004; KIM; ZIMMERMANN; NAGAPPAN, 2012; BAVOTA *et al.*, 2015): combating software design degradation, reducing the effort to perform maintenance activities, facilitating the implementation of new features, correcting bugs, and removing anomalous structures such as smells from the code. Regardless of the refactoring objective, an expectation is that refactorings contribute to improving the resulting code structure without altering its behavior (OPDYKE, 1992; FOWLER, 1999; DEURSEN *et al.*, 2001).

From the perspective of refactoring, test smells are code fragments that suggest the possibility of refactoring test codes (DEURSEN *et al.*, 2001). Most integrated develop-

Table 2.1: Characteristics of detection and refactoring tools to handle test smells (extended from Aljedaani *et al.* (2021)).

Tool	Framework	Precision	Detection technique	Interface	#Smells
DARTS†	JUnit	76%	Information Retrieval	IntelliJ plugin	3
DrTest	SUnit	UKN	Rule, Dynamic Tainting	Pharo plugin	1
DTDetector	JUnit	UKN	Dynamic Tainting	Command line	1
ElectricTest	JUnit	UKN	Dynamic Tainting	Command line	1
JNose Test	JUnit	91-100%	Rule	Web application	21
JTDog⋆	JUnit	UKN	Dynamic Tainting	IntelliJ plugin	10
Neutrino† ⋆	JUnit	UKN	Rule	Eclipse plugin	7
OraclePolish	JUnit	UKN	Dynamic Tainting	Command line	2
PyNose⋆	Unittest	UKN	Rule	Command line	17
PyTest-Smell⋆	PyTest	UKN	Rule	Command line	17
PolDet	JUnit	UKN	Dynamic Tainting	UKN	1
PraDeT	JUnit	UKN	Dynamic Tainting	Command line	1
RAIDE†	JUnit	UKN	Rule	Eclipse plugin	2
RTj†	JUnit	UKN	Rule, Dynamic Tainting	Command line	1
SoCRATES	Scala	98.94	Rule	IntelliJ plugin	6
Taste	JUnit	57%-75%	Information Retrieval	UNK	3
TeCReVis	JUnit	UKN	Metrics, Dynamic Tainting	Eclipse plugin	1
TEDD	JUnit	80%	Information Retrieval	Command line	1
TeReDetect	JUnit	UKN	Metrics, Dynamic Tainting	Eclipse plugin	1
TestEvoHound	JUnit, TestNG	UKN	Metrics	UKN	6
TestHound†	JUnit, TestNG	UKN	Metrics	Desktop	6
TestLint	Sunit	UKN	Rule, Dynamic Tainting	UKN	27
TestQ	CppUnit, JUnit	UKN	Metrics	Desktop	12
TRex†	TTCN-3	UKN	Rule	Eclipse plugin	38
tsDetect	JUnit	85%-100%	Rule	Command line	19
Unnamed	JUnit	88%	Rule	Command line	9
VITRuM⋆	JUnit	85%-100%	Rule	IntelliJ plugin	7
TSVizz⋆	JUnit	UKN	Rule	Standalone	19
MeteoR†⋆	JUnit	UKN	Dynamic Tainting	Eclipse plugin	1
SniffTest⋆	JUnit	96%, 97%	Rules/Heuristic	GUI	5
TESTAXE†⋆	JUnit	100%	Rules	Command line	5

† Refactoring support, ⋆Tool released after the publication of the mapping study

ment environments provide (semi-)automatic refactoring derived from Fowler's catalog to support developers refactoring the test code (e.g., Eclipse IDE[5] and IntelliJ IDEA[6]). Although some refactorings are common to the production and test code, the test code can require specific refactorings for fixing test smells (DEURSEN *et al.*, 2001). In this direction, some studies have proposed strategies and tools to support test code refactoring (ALJEDAANI *et al.*, 2021).

The remaining subsections present the base refactorings from Fowler's catalog, refac-

[5]Available at: https://www.eclipse.org/ide/
[6]Available at: https://www.jetbrains.com/idea/

torings specific to test code, and the approaches to perform such refactorings.

2.3.1 Refactoring operations

Each refactoring consists of one or more atomic code transformations, i.e., refactoring operations (OPDYKE, 1992; FOWLER, 1999). Opdyke (1992) proposed a set of 25 language-independent low-level refactoring operations (and one more for C++) organized into five groups:

- **(i) Creating a program entity.** Those refactorings create one class or one class member. They encompass: 1) Creating an empty class, 2) Creating a variable, and 3) Creating a function.
- **(ii) Deleting a program entity.** Those refactorings delete an unreferenced class or an unreferenced or redundant class member. They encompass: 4) Delete unreferenced class, 5) Delete unreferenced variable, and 6) Delete functions.
- **(iii) Changing a program entity.** Those refactorings change the name of one class or the attributes of one class member. They encompass: 7) Change class name, 8) Change variable name, 9) Change function name, 10) Change type, 11) Change access control mode, 12) Add function argument, 13) Delete function argument, 14) Reorder function arguments, 15) Add function body, 16) Delete function body, 17) Convert variable references to function calls, 18) Replace statement list with function call, 19) Inline function call, and 20) Change superclass.
- **(iv) Moving a variable.** Those refactorings move a variable to a superclass or subclass. They encompass: 21) Move variable to superclass, and 22) Move variable to subclasses.
- **(v) Composite refactorings.** Those refactorings are built on top of the primitive refactorings to perform more powerful operations. They encompass: 23) Abstract access to variable, 24) Convert code segment to function, and 25) Move class.

The low-level refactoring operations are the most primitive refactorings to create, delete, change, and move object-oriented entities. The `Composite refactorings` category is less primitive and supports slightly more powerful refactoring operations. The low-level refactorings support the high-level ones. Concerning the refactorings for the production code, Opdyke (1992) proposed three high-level refactorings:

- **Creating an Abstract Superclass.** Define an abstract superclass for a set of concrete classes and migrate the common behavior to an abstract superclass;
- **Subclassing and Simplifying Conditionals.** Define subclasses corresponding to the cases and migrate case-specific behavior down to the subclasses;
- **Aggregations & Reusable Components.** An aggregate object comprises components stored in the object as member variables. Refactorings involving aggregate objects and their components require that a component object be exclusive to one aggregate object.

Later, Fowler (1999) extended Opdyke's (1992) catalog with 72 low and high-level refactorings. Concerning the refactorings for the test code, Deursen *et al.* (2001) adapted and extended Fowler's (1999) catalog for fixing test smells. We describe them as follows.

Test refactorings. Those refactorings are built on top of the primitive refactorings to perform test refactorings.

- **Inline Resource.** Set up a fixture in the test code that holds the contents of the external resource. This fixture is then used instead of the resource to run the test;
- **Setup External Resource.** Make sure that the test using external resources explicitly creates or allocates them;
- **Make Resource Unique.** Use unique identifiers for all allocated resources, including a time stamp;
- **Reduce Data.** Minimize the data set up in fixtures to the bare essentials;
- **Add Assertion Explanation.** Add an explanatory message in the optional first argument of the assertions in the JUnit framework;
- **Introduce Equality Method.** Add an implementation for the "equals" method to verify the necessity of checking the equality of the object;
- **Parameterize Test.** Remove duplicate code using the PARAMETERIZED TEST annotation to define a variety of arguments.

2.3.2 Refactoring approaches

According to Fowler (1999), refactoring consists of two distinct steps: i) detecting the location to refactor the code and ii) identifying which refactoring should be applied. We can carry out those steps through manual or automated approaches.

Manual refactoring. Fowler (1999) and Deursen *et al.* (2001) provided a list of smells and possible test refactorings for fixing them in the production and test code, respectively. According to Fowler (1999), the most reliable approach to manually detecting smells and refactoring the test code is through code reviews. Manual approaches for detecting smells require a great human effort to interpret and analyze the code; thus, we can use them in smaller test suites. Another issue is that manual detection is highly subjective, relying on the developers' experience and knowledge of the software and domain. Concerning manual refactorings, Opdyke (1992) proposed the definition of pre and post-conditions to preserve the software behavior when applying refactorings. Besides finding the best refactoring to fix the smell, the developers must ensure keeping the test code behavior.

Automated refactoring. Refactoring recommendation solves two problems in automatic software refactoring (MENS; TOURWE, 2004): i) identification of refactoring opportunities and ii) selection of the correct refactoring. Many tools detect test smells and recognize them as a refactoring opportunity. However, only a few tools support developers with automatic refactoring. The RTj, DARTS, and TRex tools provide a complete solution for refactoring the detected smells. Developers should accept the entire refactoring solution, i.e., the tools do not provide the flexibility to adapt the suggested solution. The TestHound tool provides textual information (GREILER; DEURSEN; STOREY, 2013), and the RAIDE tool provides semi-automated test refactoring to fix test smells (SANTANA *et al.*, 2020). Developers should analyze whether and how to apply complementary refactorings to fix test smells. While automation is important, it is essential to understand the points at which human oversight, intervention, and decision-making should impact automation. Human developers might reject changes made by any automated programming technique, especially if they feel they have little control; there will be a natural reluctance to trust and use the automated refactoring tool (MURPHY-HILL; PARNIN; BLACK, 2011).

2.4 TEST CODE QUALITY ASSESSMENT

Software metrics offer quantitative evidence that aids software engineers in comprehending the impact of code structure on various quality attributes such as reusability, maintainability, and testability (TEMPERO *et al.*, 2010). This evidence supports the processes of planning, creating, and evaluating software. Although the presence of smells in test code may indeed influence its quality, particularly in terms of comprehension and readability (BAVOTA *et al.*, 2015; TUFANO *et al.*, 2016), other structural and process metrics can provide a comprehensive overview of test quality.

According to Pecorelli *et al.* (2020b), a valuable approach to identifying test code in need of maintenance operations is to evaluate the overall quality of test suites and their alignment with good practices in the object-oriented paradigm. In particular, throughout this study, we utilize five metrics related to test code size, complexity, and coupling. While many automated tools are available for calculating structural metrics from the source code (e.g., Eclipse Metrics and CK Metrics), we opted to utilize VITRuM (VIsualization of Test-Related Metrics), a plug-in specifically designed to calculate and visualize test-related metrics (PECORELLI *et al.*, 2020b) (refer to Table 2.2).

Table 2.2: Description of quality metrics (PECORELLI *et al.*, 2020b)

Acronym	**Quality Metrics**	**Description**
LOC	Number of Lines	Counts the number of lines
NOM	Number of Methods	Counts the number of methods
WMC	Weight Method Class	Counts the number of branch instructions in a class
RFC	Response for a Class	Counts the number of method invocations in a class
AsD	Assertion density	Percentage of assert statements concerning the total number of statements in a test class

Another factor influencing test code quality is developers' experience and roles within the project (CAMPOS *et al.*, 2023; CAMPOS; MARTINS; MACHADO, 2023). To capture aspects of the development process rather than aspects of the code itself, we analyze several ad-hoc metrics extracted from open-source repositories to characterize developers' experience and activities in the test code throughout this study. Specifically, we utilize PyDriller (SPADINI; ANICHE; BACCHELLI, 2018), a Python framework for mining software repositories, to gather information on commits, developers, modifications, diffs, and source code (refer to Table 2.3).

Table 2.3: Description of process metrics (SPADINI; ANICHE; BACCHELLI, 2018)

Acronym	Quality Metrics	Description
CCT	Code Churn Total	It measures the total number of lines of code that have been modified across the analyzed commits
CCM	Code Churn Max	It represents the maximum amount of code churn for a file in a single commit
CCA	Code Churn Average	It measures the average code churn per commit
Co	Commits Count	It measures the number of commits made to a file
Con	Contributors Count	It counts the number of developers that contributed to a file
MCon	Minor Contributors Count	It counts the number of contributors that contributed with less than 5% to the file
ConE	Contributors Experience	It measures the percentages of the lines authored by the highest contributor of a file
ALC	Lines Added Count	It counts the total lines added across the commits
ALM	Lines Added Max	It represents the maximum number of added lines for a file in a single commit
ALA	Lines Added Average	It measures the average number of lines added across the commit
RLC	Lines Removed Count	It counts the total lines removed across the commits
RLM	Lines Removed Max	It represents the maximum number of removed lines for a file in a single commit
RLA	Lines Removed Average	It measures the average number of lines removed across the commits

2.5 MACHINE LEARNING

We can categorize the ML techniques as supervised, unsupervised, self-supervised, and reinforcement learning (KAELBLING; LITTMAN; MOORE, 1996; SARKER *et al.*, 2020). *Supervised learning* is typically the task of ML to learn a function that maps an input to an output based on sample input-output pairs (HAN; KAMBER; PEI, 2011). *Unsupervised learning* analyzes unlabeled datasets without the need for human interference, i.e., a data-driven process (HAN; KAMBER; PEI, 2011). We can define *self-supervised learning* as hybridizing supervised and unsupervised algorithms because it operates on labeled and unlabeled data (SARKER *et al.*, 2020). Finally, *reinforcement learning* enables software agents and machines to automatically evaluate the optimal behavior in a particular context to improve efficiency (KAELBLING; LITTMAN; MOORE, 1996).

2.5.1 Supervised Machine Learning algorithms

This section explores supervised ML algorithms used throughout this thesis to perform classification tasks. As the literature on test refactoring classification is embryonic, we

took this opportunity to benchmark learning algorithms with different characteristics. They are: 1) Decision Trees (DT), 2) Random Forest (RF), 3) Extra-Tree (ExT), 4) Logistic Regression (LR), 5) Naive Bayes (NB), and 6) Support Vector Machine (SVM). In order to show how the algorithms work, please consider the dataset of Table 2.4 with two features related to test code quality and the class label indicating whether the test class was refactored.

Table 2.4: Dummy dataset for test class refactorings.

ID	Class Name	Complexity	Test Smells	Refactored
I0	UserServiceTest	Low	True	Yes
I1	DatabaseConnectionTest	Medium	False	No
I2	GUIControllerTest	High	True	Yes
I3	FileParserTest	Low	False	No
I4	ReportGeneratorTest	High	True	Yes
I5	DataValidatorTest	Medium	False	No
I6	LoggingManagerTest	Low	True	Yes
I7	PaymentProcessorTest	High	True	No
I8	ConfigurationReaderTest	Medium	False	Yes
I9	EmailSenderTest	High	True	No

(1) Decision Tree (DT). The DT algorithm approximates robust discrete functions in noise and can express disjunctive learning (QUINLAN, 1993; FREUND; MASON, 1999). In more detail, each node of a DT algorithm specifies a test of some attribute, and the branch descending specifies the possible values for the attribute. Therefore, a DT algorithm classifies instances by sorting them down the tree until a leaf node (MITCHELL, 1997).

While multiple ways exist to select the best attribute at each node, the Information Gain (IG) is a popular splitting criterion for decision tree models. Entropy (E) is a concept that stems from information theory, which measures the impurity of the sample values. The Entropy is given by the equation:

$$E(S) = -\sum_{c \in C} p(c) log_2 p(c) \tag{2.1}$$

where S is the dataset to calculate the entropy, c represents the classes in S, $p(c)$ represents the proportion of data points that belong to class c to the total of data points.

Information Gain represents the difference in entropy before and after a split on a given attribute. The attribute with the highest information gain will produce the

best split as it is doing the best job classifying the training data according to its target classification. The Information Gain is given by the equation:

$$IG(S,a) = E(S) - \sum_{v \in values(a)} \frac{|S_v|}{S} E(S_v) \tag{2.2}$$

where a represents an attribute or class, $E(S)$ is the entropy of a dataset S, $|S_v|/|S|$ represents the proportion of the values in S_v to the number of values in S, and $E(S_v)$ is the entropy of the S_v dataset.

Now, let us calculate the Entropy and Information Gain for each feature in the dataset in Table 2.4 and determine the best feature to split on. For simplicity, we use a binary split for each feature.

$$\begin{aligned} E(S) &= -p(\text{Refactored}) \cdot \log_2(p(\text{Refactored})) - p(\text{NotRefactored}) \cdot \log_2(p(\text{NotRefactored})) \\ &= -\frac{5}{10} \cdot \log_2\left(\frac{5}{10}\right) - \frac{5}{10} \cdot \log_2\left(\frac{5}{10}\right) \\ &= 1 \end{aligned} \tag{2.3}$$

Then, we calculate the information gain for the complexity subset $IG(Complexity, S)$:

$$\begin{aligned} IG(\text{Complexity}, S) &= E(S) - \sum_{v \in \{\text{Low,Medium,High}\}} \frac{|S_v|}{|S|} E(S_v) \\ &= 1 - \left(\frac{3}{10} E(S_{\text{Low}}) + \frac{3}{10} E(S_{\text{Medium}}) + \frac{4}{10} E(S_{\text{High}})\right) \end{aligned} \tag{2.4}$$

where S_{Low}, S_{Medium}, and S_{High} are the subsets of the dataset where *Complexity* is *Low*, *Medium*, and *High*, respectively. When calculating their entropy, we have:

- $E(S_{Low}) = -\frac{3}{3} \cdot \log_2\left(\frac{3}{3}\right) - 0 \cdot \log_2(0) = 0$
- $E(S_{Medium}) = -\frac{0}{3} \cdot \log_2\left(\frac{0}{3}\right) - \frac{3}{3} \cdot \log_2\left(-\frac{3}{3}\right) = 0$
- $E(S_{High}) = -\frac{3}{4} \cdot \log_2\left(\frac{3}{4}\right) - \frac{1}{4} \cdot \log_2\left(-\frac{1}{4}\right) \approx 0.811$

Replacing those values in the Equation 2.4, we have:

$$IG(\text{Complexity}, S) = 1 - \left(\frac{3}{10} \cdot 0 + \frac{3}{10} \cdot 0 + \frac{4}{10} \cdot 0.811\right) \approx 0.317 \tag{2.5}$$

Similarly, we calculate the information gain for $IG(TestSmells, S)$, resulting in:

$$\begin{aligned} IG(\text{Code Smells}, S) &= E(S) - \sum_{v \in \{\text{True,False}\}} \frac{|S_v|}{|S|} E(S_v) \\ &= 1 - \left(\frac{6}{10} E(S_{\text{True}}) + \frac{4}{10} E(S_{\text{False}})\right) \\ &= 1 - \left(\frac{6}{10} \cdot 0.65) + \frac{4}{10} \cdot 0\right) \approx 0.369 \end{aligned} \tag{2.6}$$

where D_{True} and D_{False} are the subsets of the dataset where Test Smells is True and False, respectively.

Subsequently, we choose the feature with the highest information gain as the root node of the decision tree. In this case, it's *Complexity*. Then, we continue the process recursively for each branch until we build the tree in Figure 2.3.

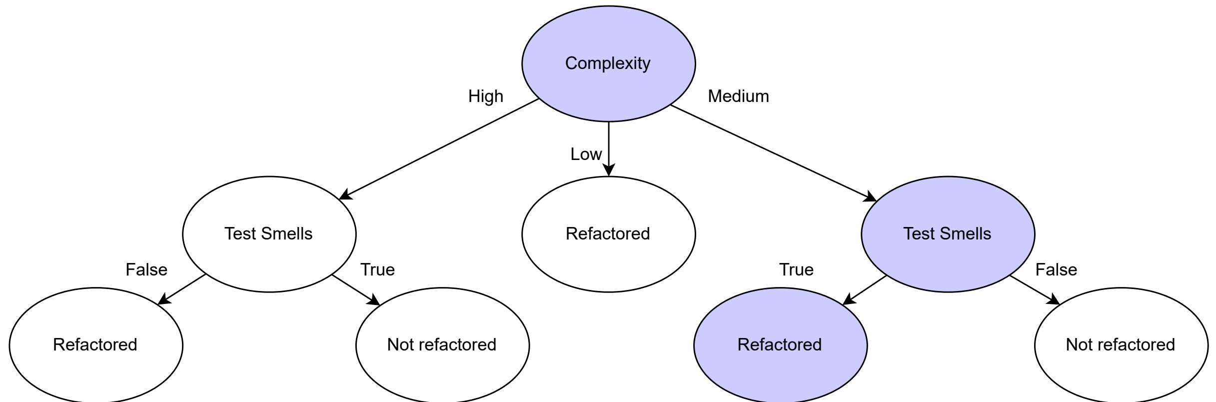


Figure 2.3: Building the Decision Tree for the Test Refactoring classification. The circles highlighted in blue represent the steps to classify the new instance ReceiverEmailTest.

Finally, we can classify the new instance ⟨EmailReceiverTest, Medium, True⟩ by comparing the values with the tree nodes and branches. In particular, we follow the branch *Medium* below *Complexity*. Then, we follow the branch *True* under *Medium*. Therefore, the classification for the instance is *Refactored*.

(2) Random Forest (RF). The RF algorithm contains trees based on the values of a random subset of instances experimented individually and with the same distribution for all trees in the forest (HO, 1995; BREIMAN, 2001). The RF algorithm's key idea is to introduce randomness in data selection and the features considered during tree-building.

While the DT algorithm is based upon a fixed set of attributes and often overfit, RF uses the *bagging* technique to build full DTs in parallel from random bootstrap samples of the dataset. The representation of bagging for the RF algorithm is given by:

$$Bagging : D_i^{'} = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) \tag{2.7}$$

where $D_i^{'}$ is the randomly sampled subset of the D original dataset.

In addition to using a random subset of instances, each DT in the RF algorithm considers only a random subset of features at each split. This randomness helps in decorating the trees and improves the diversity among them. With this, the RF algorithm combines the predictions of trees through voting for classification tasks, providing a more robust and accurate model compared to individual DTs. As a result, it helps overcome overfitting associated with individual DTs.

In each iteration of building a decision tree within the RF algorithm, a random subset of instances is selected with replacement (bootstrapping). For example, let us consider RF with three trees with the following instances randomly selected from Table 2.4:

- **Random Forest #1:** with the instances [I7, I6, I4, I0, I6, I6, I8, I2, I8, I0]
- **Random Forest #2:** with the instances [I3, I8 I5, I2, I0, I3, I8, I2, I8, I6]
- **Random Forest #3:** with the instances [I3, I2, I9, I4, I4, I2, I8, I3, I4, I3]

Figure 2.4 presents the results for the Entropy at each node of the decision trees built with the RF algorithm. To classify the new instance ⟨EmailReceiverTest, Medium, True⟩, we can consider the categories *Low*, *Medium*, and *High* from the categorical variable *Complexity* as numerals 0, 1, and 2, respectively. Similarly, we consider the *True* values for test smells as 1. Therefore, in *Random Forest #1*, we classify the new instance as *Refactored* because of the *Complexity*. In *Random Forest #2*, we follow the right branch below *Test Smells*, as the new instance contains smells. In *Random Forest #3*, we follow

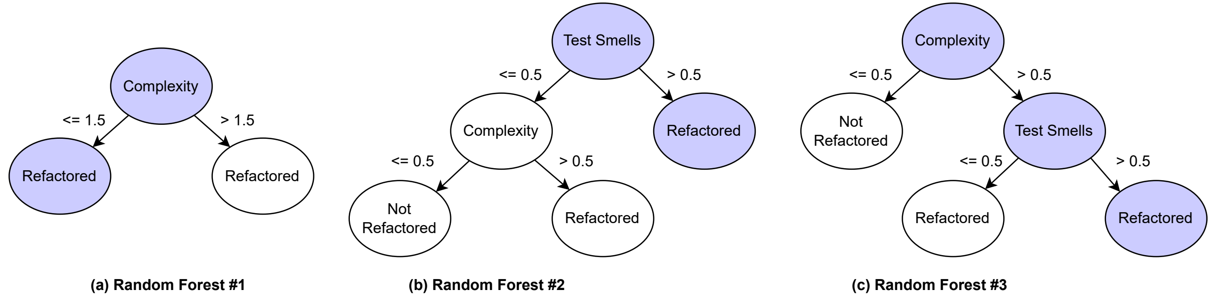


Figure 2.4: Building the Trees with Random Forest algorithm for the Test Refactoring classification.

to the right branch below *Complexity*, as the new instance has *Medium* complexity (1). Then, we follow to the right branch below *Test Smells*, and classify the instance as *Refactored*.

(3) Extra-Tree (ExT). ExT is another ensemble learning method based on the RF algorithm. It introduces additional randomness during rebuilding, making it even more robust to overfitting. In particular, ExT algorithm uses random subsets of instances (bagging) and random subsets of features and selects split points at random rather than searching for the optimal split (GEURTS; ERNST; WEHENKEL, 2006). Overall, the added randomness in ExT algorithm enhances diversity, reduces computational costs as the algorithm does not need to search for optimal split points, and helps improve the performance generalization of the model.

In each iteration of building a decision tree within the ExT algorithm, a random subset of instances is selected with replacement as well as the features (*Test Smells*, *Complexity*). For example, let us consider ExT with three trees with the following features and instances randomly selected from Table 2.4:

- **Extra-Tree #1:** with the instances [I6, I4, I9, I5, I4, I8, I5, I7, I1, I8], and features [Complexity, Test Smells].
- **Extra-Tree #2:** with the instances [I3, I2, I9, I4, I7, I3, I0, I6, I7, I2], and features [Complexity, Test Smells].
- **Extra-Tree #3:** with the instances [I6, I9, I0, I9, I4, I9, I1, I5, I0, I2], and features [Complexity].

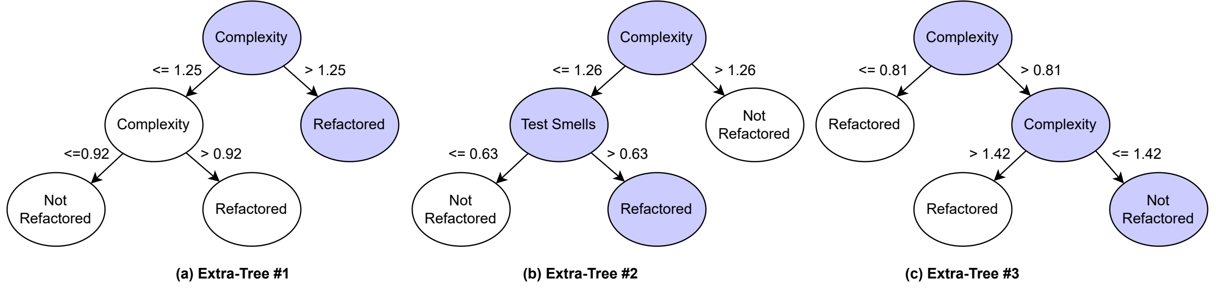


Figure 2.5: Building the Trees with ExT algorithm for the Test Refactoring classification.

Figure 2.5 presents the decision trees built with the ExT algorithm by calculating the *Entropy* and *Info-Gain* as discussed previously. In order to classify the new instance ⟨EmailReceiverTest, Medium, True⟩, we can consider the categories *Low*, *Medium*, and *High* from the categorical variable *Complexity* as numerals 0, 1, and 2, respectively. Similarly, we consider the *True* values for test smells as 1. Therefore, in *Extra-Tree #1*, we follow the left branch below *Complexity*, as the new instance has *Medium* complexity $(1 < 1.99)$. Then, we follow the right branch below *Test Smells* as the new instance is smelly, classifying the new instance as *Refactored*. In *Extra-Tree #2*, we follow the left branch below *Complexity*, then the right branch below *Test Smells*, classifying the new instance as *Refactored*. In *Extra-Tree #3*, we follow the left branch below *Complexity* twice and classify the new instance as *Not Refactored*. As the ExT algorithm decides the new instance class by voting, 2 out of 3 trees classified the instance as *Refactored*, which is the final label.

(4) Logistic Regression (LR). LR algorithm uses linear functions as $f : X \rightarrow Y$ or $P(Y|X)$ where Y is a discrete-valued, and $X = \langle X_1 ... X_n \rangle$ is any vector containing discrete or continuous variables to establish a correlation between the unobserved and observed instances (MITCHELL, 1997; BISHOP, 2006). In particular, we considered Y as a boolean variable, e.g., for classifying whether a test class refactored $(Y = 1)$ or not refactored $(Y = 0)$. The model assumed by LR algorithm where Y is boolean is:

$$
\begin{aligned}
P(Y=0|X) &= \frac{exp(w_0 + \sum_{i=1}^{n} w_i X_i)}{1 + exp(w_0 + \sum_{i=1}^{n} w_i X_i)} \\
P(Y=1|X) &= \frac{1}{1 + exp(w_0 + \sum_{i=1}^{n} w_i X_i)}
\end{aligned}
\tag{2.8}
$$

where *exp* models the odds ratio using the exponential function of the linear combination of the coefficients and independent variables, w refers to the log-odds of the probability of the dependent variable being 1 when all independent variables are 0 (w_0) and when each independent variable is 0 (w_i).

The probability transformed into odds using the sigmoid function:

$$Y = \frac{1}{1 + exp(-X)} \tag{2.9}$$

Now, considering the dataset of test code refactorings in Table 2.4, we have to find the vector of weights w. Typically, LR model involves an optimization algorithm, such as gradient descent, to find the optimal values for the weights that minimize a cost function. Therefore, we will not calculate them here but rely on the algorithm execution. As an output, the algorithm returns the intercept ($w_0 = -0.39$), and the weights ($w_1 = 0.01 w_2 = 0.63$). Therefore, we have:

$$\begin{aligned} P(Y=0|X) &= \frac{exp(-(0.39 + 0.01 \cdot Complexity + 0.63 \cdot TestSmells))}{1 + exp(-(0.39 + 0.01 \cdot Complexity + 0.63 \cdot TestSmells))} \\ P(Y=1|X) &= \frac{1}{1 + exp(-(0.39 + 0.01 \cdot Complexity + 0.63 \cdot TestSmells))} \end{aligned} \tag{2.10}$$

Now, let us classify the new instance ⟨EmailReceiverTest, Medium, True⟩. We can transform the categorical into numerical variables. As a result, the values *Low*, *Medium*, *High* are converted into 0, 1, and 2 for the *Complexity* feature. Similarly, the value *True* for the *Test Smells* feature is 1, and *False* is 0. It means that the new instance has $Complexity = 1$ and $TestSmells = 1$, resulting in:

$$\begin{aligned} P(Y=0|X) &= \frac{exp(-(0.39 + 0.01 \cdot 1 + 0.63 \cdot 1))}{1 + exp(-(0.39 + 0.01 \cdot 1 + 0.63 \cdot 1))} \\ P(Y=1|X) &= \frac{1}{1 + exp(-(0.39 + 0.01 \cdot 1 + 0.63 \cdot 1))} \end{aligned} \tag{2.11}$$

$$\begin{aligned} P(Y=0|X) &\approx 0.26 \\ P(Y=1|X) &=\approx 0.73 \end{aligned} \tag{2.12}$$

As $P(Y = 1|X) \approx 0.73$, we classify the new instance as *Refactored*. Figure 2.6 shows the distribution of data points in the feature space (yellow points for *Refactored* classes), the new instance classification (red x marker), and the decision boundary to indicate predicted probabilities of a class being *Refactored* (superior area).

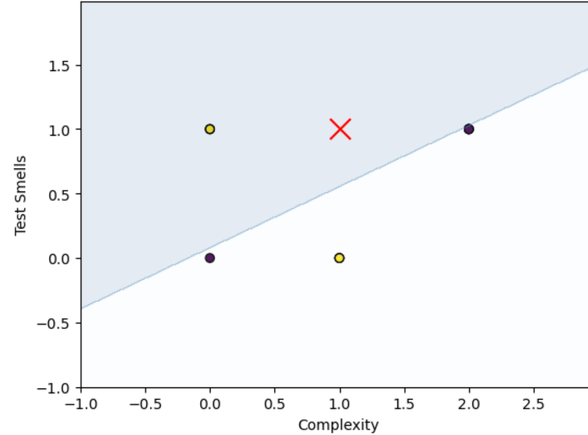


Figure 2.6: Classification of the new instance *EmailReceiverTest* using LR algorithm.

(5) Naive Bayes (NB). The NB algorithm is a classification algorithm based on Bayes' rule and conditional independence assumptions (MITCHELL, 1997; BISHOP, 2006). Considering a classification task with Y representing a binary class of refactored and non-refactored test classes and a set of features $X = \langle X_1 ... X_n \rangle$, the goal is to predict the probability of a class refactored given the observed features. The NB algorithm calculates that probability using Bayes' theorem:

$$P(Y|X_1, X_2, \cdots, X_n) = \frac{P(Y) \cdot P(X_1|Y) \cdot P(X_2|Y) \cdot \cdots \cdot P(X_n|Y)}{P(X_1) \cdot P(X_2) \cdot \cdots \cdot P(X_n)} \tag{2.13}$$

where $P(Y)$ is the prior probability of a class refactored, $P(X_1) \cdot P(X_2) \cdot \cdots \cdot P(X_n)$ serves as a normalization constant, and $P(X_i|Y)$ is the likelihood of feature X_i given a class refactored $Y = 1$, and $P(Y|X_1, X_2, \cdots, X_n)$ is the posterior probability of a class refactored.

The naive assumption is that features are conditionally independent given the class label, simplifying the calculation, as it allows us to model the likelihood of each feature independently:

$$P(X_i|Y) = P(X_i|Y, X_1, X_2, \cdots, X_{i-1}, X_{i+1}, \cdots, X_n) \tag{2.14}$$

Under this assumption, the formula simplifies to:

$$P(X_1...X_n|Y) = \prod_{i=1}^{n} P(X_i|Y) \tag{2.15}$$

Now, let us consider a practical example using a dataset with features representing the complexity, the presence of test smells, and the class being refactored (Table 2.4). Supposing we want to classify the new class ⟨EmailReceiverTest, Medium, True⟩ using the NB algorithm, we first have to calculate the number of instances as:

- Total number of instances (N): 10
- Number of refactored instances (Y): 5
- Number of not refactored instances (N): 5
- Number of refactored instances with Medium Complexity ($M|Y$): 1
- Number of not refactored instances with Medium Complexity ($M|N$): 2
- Number of refactored instances with Code Smells = True ($T|Y$): 4
- Number of not refactored instances with Code Smells = True ($T|N$): 1

We calculate the probabilities considering each feature:

- $P(Y) = \frac{Y}{N} = \frac{5}{10} = 0.5$
- $P(N) = \frac{N}{N} = \frac{5}{10} = 0.5$
- $P(M|Y) = \frac{M|Y}{Y} = \frac{1}{5} = 0.2$
- $P(T|Y) = \frac{T|Y}{Y} = \frac{4}{5} = 0.8$
- $P(M|N) = \frac{M|N}{N} = \frac{2}{5} = 0.4$
- $P(T|N) = \frac{T|N}{N} = \frac{1}{5} = 0.2$

Then, we calculate the probability for the new instance being refactored using Bayes' Theorem:

$$P(Y|M,T) = \frac{P(M|Y) \cdot P(T|Y) \cdot P(Y)}{P(M|Y) \cdot P(T|Y) \cdot P(Y) + P(M|N) \cdot P(T|N) \cdot P(N)} \tag{2.16}$$

$$P(Y|M,T) = \frac{0.2 \cdot 0.8 \cdot 0.5}{0.2 \cdot 0.8 \cdot 0.5 + 0.4 \cdot 0.2 \cdot 0.5} = \frac{0.08}{0.08 + 0.04} = \frac{0.08}{0.12} \approx 0.67 \tag{2.17}$$

Therefore, the probability that the new instance is refactored is approximately 0.67, and the probability that it is not refactored is $1 - 0.67 = 0.33$. Consequently, we classify the new instance as *Refactored*.

(6) Support Vector Machines (SVM). SVM algorithm computes a hyper-plane in high-dimensional space to classify data into predefined classes. The algorithm searches for the best hyperplane for error minimization and geometric margin maximization (CORTES; VAPNIK, 1995). While SVM algorithm is versatile enough to handle linear and nonlinear classification tasks, kernel functions become essential when dealing with non-linearly separable data (NOBLE, 2006).

In the case of Linear SVM algorithm, it creates a hyperplane separating the linearly separable data into classes. The hyperplane is defined by a weight vector w and a bias term b. The decision function for a binary classification problem is given by:

$$\hat{y} = \begin{cases} 1 : w^T x + b \geq 0 \\ 0 : w^T x + b < 0 \end{cases} \tag{2.18}$$

where x represents the input feature vector, w is the weight vector, b is the bias term, and $\hat{y}$ is the predicted class.

Supposing that we want to classify a new instance ⟨EmailReceiverTest, Medium, True⟩, denoted as input vector x, based on the dataset in Table 2.4. We first need to encode the categorical feature *Complexity* into numerical values, 0, 1, and 2, for the categories *Low*, *Medium*, and *High*. So, for the new instance, the encoded feature vector is $x = \langle 1, 1 \rangle$.

Subsequently, we use the dataset to calculate the weight vector w:

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i \tag{2.19}$$

However, we need to find the Lagrange multipliers (α_i) for the support vectors in the dataset. The Lagrange multipliers involve solving a quadratic optimization problem, which is typically done using optimization algorithms such as Sequential Minimal Optimization (SMO); therefore, we will not calculate them here but rely on the algorithm execution. Considering the support vectors

- `DatabaseConnection`, encoded as [2, 0], has $\alpha = -0.2$
- `FileParser`, encoded as [0, 1], has $\alpha = -1.0$
- `DataValidator`, encoded as [1, 0], has $\alpha = -1.0$
- `PaymentProcessor` encoded as [1, 0], has $\alpha = -1.0$
- `EmailSender`, encoded as [1, 0], has $\alpha = -1.0$
- `UserService`, encoded as [0, 1], has $\alpha = -1.0$
- `GUIController`, encoded as [2, 1], has $\alpha = 0.2$
- `ReportGenerator`, encoded as [2, 1], has $\alpha = -1.0$
- `LoggingManager`, encoded as [0, 1], has $\alpha = -1.0$
- `ConfigurationReader`, encoded as [1, 0], has $\alpha = -1.0$

Now, we replace the values in the equation 2.19, as follows:

$$\begin{aligned} w = -0.2 \times 1 \times \langle 2,0\rangle + (-1.0) \times 1 \times \langle 0,0\rangle + (-1.0) \times 1 \times \langle 1,0\rangle \\ +(-1.0) \times 1 \times \langle 2,1\rangle + (-1.0) \times 1 \times \langle 2,1\rangle + 0.2 \times 1 \times \langle 0,1\rangle \\ +1.0 \times 1 \times \langle 2,1\rangle + 1.0 \times 1 \times \langle 2,1\rangle + 1.0 \times 1 \times \langle 0,1\rangle \\ +1.0 \times 1 \times \langle 1,0\rangle = \langle -0.4, 1.2\rangle \end{aligned} \tag{2.20}$$

Therefore, the weight vector w is $\langle -0.4, 1.2\rangle$. This vector defines the normal to the hyperplane that separates the classes. The bias term b can be obtained from any support

vector. let us use the first support vector

$$b = y_1 - w \cdot x_1 b = 1 - \langle -0.4, 1.2 \rangle \cdot \langle 2, 0 \rangle = 1.8 \tag{2.21}$$

Now, we can use this w to classify the instance $\langle EmailReceiverTest, Medium, True \rangle$:

$$\hat{y} \begin{cases} 1 & \text{if } \langle -0.4, 1.2 \rangle \cdot \langle 1, 1 \rangle + 1.8 \geq 0 \\ 0 & \text{if } \langle -0.4, 1.2 \rangle \cdot \langle 1, 1 \rangle + 1.8 < 0 \end{cases} \tag{2.22}$$

In this case, $2.6 \geq 0$, so the new instance is classified as *Refactored*.

On the other hand, nonlinear SVM algorithm is used when the data is not linearly separable. It involves transforming the input data into a higher-dimensional feature space through kernel functions, which are classified using a hyperplane in the new space.

$$f(x) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i K(x, x_i) + b\right) \tag{2.23}$$

where N is the number of support vectors, y_i represents the class label in the support vector, $K(X, X_i)$ is the kernel function, and b is the bias.

Figure 2.7 presents a comparison among the linear SVM and Kernel functions. In particular, the Kernel functions considered to classify the new instance are:

- Radial Basis Function or Gaussian Kernel, given by $K(x, x_i) = \exp\left(-\frac{\|x - x_i\|^2}{2\sigma^2}\right)$
- Sigmoid, given by $K(x_i, x_j) = tanh(\alpha x_i T x_j + b)$

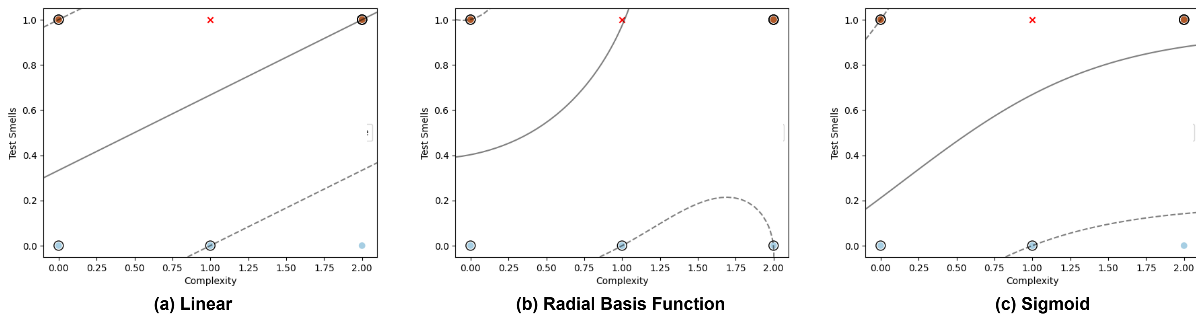


Figure 2.7: Comparison among difference kernel functions for SVM algorithm. The x highlighted in red represent the new instance ReceiverEmailTest.

Conclusion. The ML algorithms exhibit varied strengths and weaknesses based on the characteristics of the dataset to which they are applied. The DT algorithm can be prone to overfitting high-dimensional data and capturing noise, while the SVM algorithm excels in such spaces. The NB algorithm is robust in handling sparse data, particularly in text and categorical contexts. Due to their ensemble nature, the RF and ExT algorithms are adept at managing high-dimensional and sparse datasets. The LR algorithm is well-suited for linear relationships, making it effective when dealing with linear data. Figure 2.8 shows the boundaries for the ML algorithms for a classification task using a dummy dataset with two features. The dataset is relatively dense, with no redundant features.

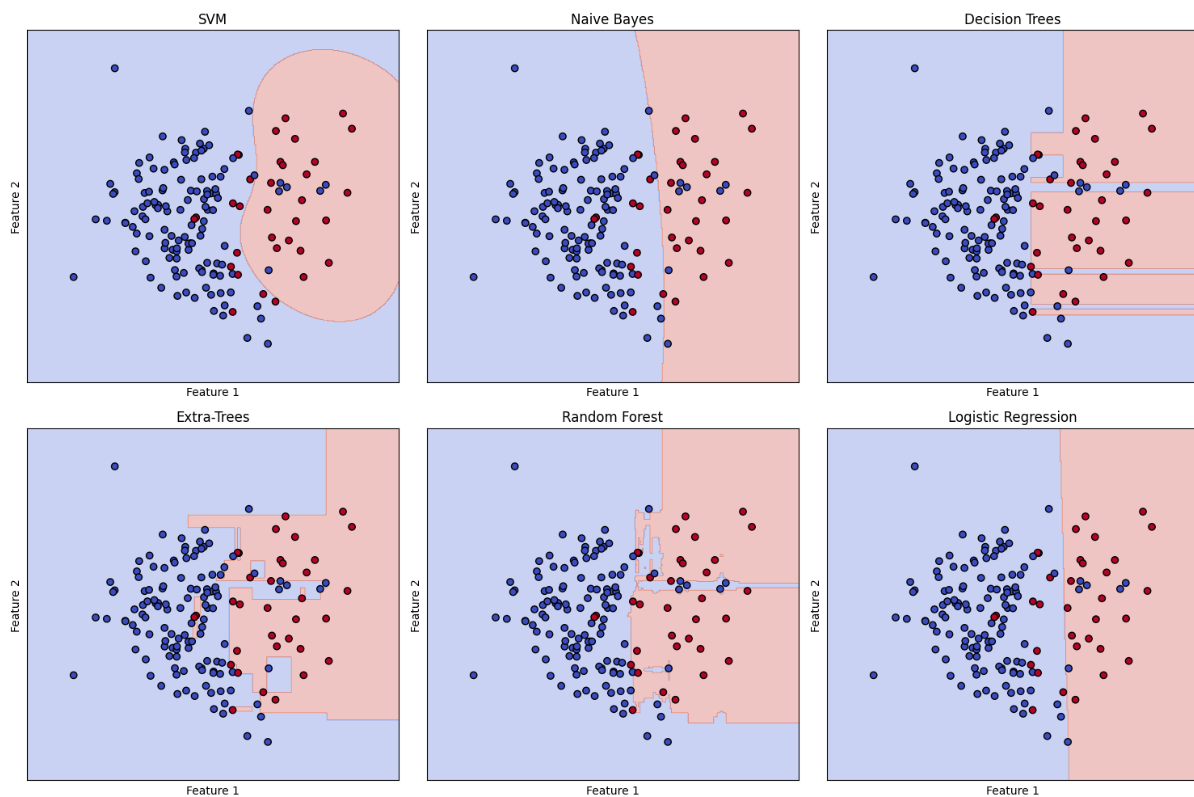


Figure 2.8: Boundaries for ML algorithms based on a binary dummy dataset, where the scattered points represent the training data points and respective classification.

2.5.2 Evaluation Metrics

We can use the *Precision*, *Recall*, *F1-Score*, and *Matthews Correlation Coefficient* (MCC) standard evaluation metrics to evaluate the performance of machine learning models. Those metrics analyze four possible outcomes from the confusion matrix in Table 2.5. To

understand the confusion matrix, consider a task of identifying which instances of the test code contain a test smell.

Table 2.5: Confusion Matrix for smell detection outcomes.

		Actual	
		Positive	Negative
Predicted	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

- **True Positive (TP).** A smelly instance is classified as smelly;
- **False Negative (FN).** A smelly instance is classified as non-smelly;
- **True Negative (TN).** A non-smelly instance is classified as non-smelly;
- **False Positive (FP).** A non-smelly instance is classified as smelly.

Therefore, the confusion matrix represents a valuable asset for evaluating the performance of an ML model. We can calculate the metrics (KUHN; JOHNSON, 2013):

- **Recall.** This metric represents how much the model detects existing smelly instances. The recall is the percentage of the true positives (TP) over the number of true positives (TP) and false negatives (FN);

$$recall = \frac{TP}{TP + FN} \tag{2.24}$$

- **Precision.** This metric represents how much the predictions of smelly instances are correct. The precision is the percentage of the true positives (TP) over the number of true positives (TP) and false positives (FP);

$$precision = \frac{TP}{TP + FP} \tag{2.25}$$

- **F1-Score.** This metric represents the harmonic mean of precision and recall. The F-Score is the weighted average of the precision and recall times two. It also refers to F1 or F-Measure.

$$f1 = 2 * \frac{precision * recall}{precision + recall} \tag{2.26}$$

- **Matthews Correlation Coefficient (MCC).** This metric considers both false positives and false negatives. Therefore, it provides a useful and balanced measure when dealing with imbalanced datasets.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.27}$$

2.5.3 Datasets for detection and refactoring of test smells

There are few publicly available datasets with information on test smells. The datasets differ regarding the test smell detection tools and domain and project versions, making it hard to compare their outcomes. The existing datasets are:

- **Detection of the presence of test smells.** Peruma *et al.* (2020a) released an accurate dataset[7] with information on test smells in 60 test classes from Android apps. The dataset contains test smells detected with the tsDetect tool, which returns a boolean value for a given test smell in the test class;
- **Detection of test smells.** Virginio *et al.* (2021) used the same dataset as Peruma *et al.* (2020a) to compare the accuracy between the JNose Test[8] and the tsDetect tools to detect test smells. Therefore, the authors expanded the dataset[9] to a finer granularity (method, block, and line level), providing the exact location and number of test smells in a test class;
- **Detection of test smells through the evolution of the test code.** Kim, Chen and Yang (2021) extended the tsDetect tool to detect test smells in a finer granularity (method and line level) and aggregated them per file. The authors created a new dataset[10] using the tool extension to detect test smell through the lifetime of 13 open-source projects.

[7] Available at: <https://testsmells.org/pages/research/experimentdata.html>
[8] Available at: https://jnosetest.github.io/
[9] Available at: <https://github.com/arieslab/JNose-Validation>
[10] Available at: <https://github.com/SPEAR-SE/TestSmellEmpirical_Data>

2.6 CHAPTER SUMMARY

Software testing is a fundamental activity for software quality assurance. Frequently, developers count on testing frameworks (e.g., JUnit) to write, organize, and execute test suites. However, developers can insert test smells while writing the test code, which can negatively affect the test code quality, harming the software testing and maintenance activities. To eliminate test smells, developers should refactor the test code in a way that does not alter test logic. This background chapter presented the basic concepts of test smells, testing frameworks, and test refactorings.

We also introduced the concepts related to techniques used to handle test smells. In particular, we focused on using ML techniques, which we consider relevant for developing and evaluating our approach to identify refactoring opportunities and suggest proper refactoring operations through this thesis.

Chapter 3

RELATED WORK

Since its inception, both interests in understanding test smells and their consequences on test and software quality have been growing and evolving rapidly (ALJEDAANI *et al.*, 2021). We know several recently published and ongoing studies on test smell that can be important to incorporate and discuss in this thesis. Therefore, we performed an ad-hoc review in cycles to cope with the rapid research development on test smells.

We conducted an ad-hoc search on Google Scholar using the string created by Garousi and Küçük (2018): `((bad OR code OR test) AND (smell OR antipattern)) AND ("test code" OR "unit test")`. Initially, we compiled a list of 54 studies up to 2020 and subsequently added more as they were published. In the second cycle, we included 21 additional studies (from 2021 to 2022), and in the third cycle, we added 20 more studies (from 2023 to 2024). In total, we synthesized 95 studies to avoid duplicate efforts and to identify further strategies, methods, techniques, and tools for addressing test smells. This review process facilitated continuous evidence monitoring.

The remainder of the chapter presents a body of knowledge on test smells that provides an understanding of the developers' perception, the impacts on the software quality, and the current practices to handle test smells. Section 3.1 presents the catalogs and literature reviews on test smells and test refactorings. Section 3.2 focuses on the literature regarding the diffusion and effects of test smells on the test code quality. Section 3.3 presents the studies of test smells in testing frameworks besides JUnit. Section 3.4 lists the tools to identify test smells and refactor test codes. Section 3.5 presents studies investigating the developers' perception of the test smells. Section **??** presents the studies

investigating the influence of the developers' experiences on the test code quality. Section 3.6 presents the literature regarding the test refactorings to fix test smells. Section 3.7 presents the usage of Machine Learning (ML) techniques to handle test smells. Section 3.8 summarizes the main limitations found in the related work.

3.1 CATALOGS AND REVIEWS ON TEST SMELLS AND REFACTORINGS

Deursen *et al.* (2001) introduced the concept of test smells to denote poorly designed test cases. The authors proposed a first catalog describing test smells and refactorings to fix them. Meszaros, Smith and Andrea (2003) broaden that concept, specifying seven test smells related to the test code level and five test smells related to the test behavior. Later on, Guerra and Fernandes (2007), Reichhart, Gîrba and Ducasse (2007), Greiler, Deursen and Storey (2013), Kummer, Nierstrasz and Lungu (2015), Peruma *et al.* (2019), Delplanque *et al.* (2019), and Yang *et al.* (2023) proposed new test smells based on test code analysis and developers' perceptions. Bowes *et al.* (2017) listed relevant best practices for tests with their possible quantification concerning test smells.

Garousi and Küçük (2018) and Garousi, Küçük and Felderer (2019) presented a multivocal literature mapping of the scientific and grey literature to analyze and classify the body of knowledge on test smells. The authors collected data from 120 sources from the industry (e.g., posts in blogs and videos) and 46 sources from academia published until April 2016. They presented an extensive set of test smells in the literature[1], including 196 test smells and 12 tools. In addition, they provided a summary of guidelines, techniques, and approaches to dealing with test smells.

Aljedaani *et al.* (2021) conducted a systematic mapping to complement the mentioned reviews regarding the available tools. The authors selected 47 peer-reviewed papers published until December 2020. Their contributions include a list of 22 tools and their comparison regarding the supported test smells, environment, and detection strategies. In summary, the tools support the detection of 66 test smells, and five tools support test refactorings to fix 10 test smells across seven different testing frameworks.

Rather than relying on preexisting catalogs, our study identifies test smells based on developers' perceptions of issues they consider significant and essential to address in real-world scenarios (RQ_1). Therefore, we could identify emerging test-specific refactorings beyond the capabilities of state-of-the-art automated tools for handling test smells.

[1]Available at: https://goo.gl/1ZrL65

3.2 INVESTIGATION OF TEST SMELLS EFFECTS ON SOFTWARE QUALITY

Several studies explored the relationship between test smells and software quality from different perspectives. Bavota *et al.* (2012) conducted two key empirical studies. The first was an exploratory study on the diffusion of nine test smells in 18 software projects. The second study was a controlled experiment with 20 participants to investigate the effects of test smells on software maintenance. Later, Bavota *et al.* (2015) extended both studies by analyzing 27 software projects with 61 participants. Both studies (BAVOTA *et al.*, 2012; BAVOTA *et al.*, 2015), had similar results. The exploratory research presented a high diffusion of test smells in JUnit test classes. The controlled experiment showed test smells could reduce test code comprehension compared to the absence of test smells. Similarly, Peruma *et al.* (2019) performed an empirical study on the distribution and survivability of test smells on 656 open-source Android apps. The results indicated the widespread occurrence of test smells in apps, which emerge early in their lifetime.

Other studies have investigated the relationship between test smells and structural metrics. Rompaey *et al.* (2007) proposed a set of metrics defined in concepts of unit tests to formalize and detect test smells. Tahir, Counsell and MacDonell (2016) conducted an exploratory study with five open-source projects to investigate the relationship between five structural metrics of production classes and nine test smells. The results indicated the complexity of the production classes is a good indicator of test smells. Pecorelli *et al.* (2020a) performed an empirical study with 1,780 open-source Android apps to assess how tested those apps are and how well-designed the tests are, considering test smells and structural metrics of the test code. Although the authors did not perform a correlation study, their results indicated the test classes have a low design quality, considering the structural metrics and test smells.

Similarly, Martins, Costa and Machado (2023) conducted an empirical study on 13,703 open-source Java systems to investigate i) the relationship between test smells and structural metrics of test code and ii) the relationship between test smells. Results indicated the Sleepy Test (ST), Mystery Guest (MG), and Resource Optimism (RO) test smell rarely occur, and the last two are strongly correlated, indicating those test smells are more severe than others. Additionally, test smells have a moderate correlation with structural metrics. Stefano *et al.* (2022) investigated the relationship between architectural and test smells from 40 open-source Java projects. As a result, the Eager Test (ET) and Assertion Roulette (AR) test smells often occur with architectural smells.

All the studies above investigated manually written tests. However, the production code quality can also influence the test code generation by automated tools. In this direction, Palomba *et al.* (2016) analyzed the diffusion of test smells in the test code of 110 open-source projects generated with the support of the Evosuite tool[2]. The results indicated that 83% of JUnit tests exhibited at least one test smell, similar to the test suites manually written. When comparing different test generation tools, Grano *et al.* (2019) studied the influence of production class properties on the generation of smelly test code using automated tools (Randoop[3],JTExpert[4], and Evosuite) in ten open-source projects. Results showed the production code size and cohesion influence the generation of smelly test classes, mainly with the Evosuite tool.

Similarly, Virginio *et al.* (2020) investigated the generated test code quality by automated test tools (Randoop and Evosuite) with the existing unit test suite of 21 open-source Java projects regarding the presence of 19 test smells. Results indicated a significant difference in test suite quality; the existing tests had a smaller distribution of test smells than those generated by tools. Panichella *et al.* (2020) investigated how effective the test smell detection tools are on automatically generated test suites for 100 Java classes (Evosuite). Unlike preceding investigations, the authors found test smells are commonly present in a small but nontrivial portion of automatically generated test suites. Later, Panichella *et al.* (2022) extended their study and compared a curated dataset with the output of the Test Smell Detector (BAVOTA *et al.*, 2012) and tsDetect (PERUMA *et al.*, 2020a) tools. Results indicated the tools are limited and misclassified over 70% of test smells, suggesting they need more appropriate metrics to match the development practices. More recently, Afonso and Campos (2023) augmented the EvoSuite tool by considering metrics of test smells. They found the number of smelly tests reduced by 3% compared to the default of the EvoSuite tool, and the tests had similar coverage and fault detection effectiveness.

Other studies explored the software quality from code coverage, defect, and change proneness perspectives. Virginio *et al.* (2019) performed an empirical study with 11 open-source projects to investigate the relationship between test coverage and 21 test smells. Their results indicated a positive relationship between test smells and test coverage. Conversely, Qusef, Elish and Binkley (2019) performed a case study with 28 versions of Apache Ant to investigate the relationship between test smells and production code

[2]<https://www.evosuite.org/>
[3]<https://randoop.github.io/randoop/>
[4]<https://sites.google.com/site/saktiabdel/JTExpert>

faults. Results indicated the number of test smells increases as the project evolves, and a positive correlation exists between some test smells and faults in the production code. Spadini *et al.* (2018) investigated ten open-source projects to find a relation between six test smells and the change and defect-proneness. They found smelly JUnit tests are more change-prone and defect-prone than non-smelly ones. Besides, the more test smells, the higher this effect, especially for the ET, AR and Indirect Testing (IdT) test smells. The production code is more defect-prone when tested by smelly tests. Wu *et al.* (2022) explored the impact of eliminating test smells on the production code quality of ten open-source projects. Results indicated refactoring the test code to fix test smells improves the code quality. The authors also identified eliminating test smells, especially the AR test smell, significantly reduces the defect- and change-proneness of the production code.

In contrast, our work extends current knowledge by assessing how test refactoring is applied and its impact on multiple aspects of test code. Specifically, we do not limit ourselves to analyzing test smells but also consider additional indicators of test code quality. In this sense, our study represents a more comprehensive analysis of the role of test refactoring (RQ_2).

3.3 INVESTIGATION OF TEST SMELLS IN DIFFERENT FRAMEWORKS

Although most of the research on test smells focused on Java programming language with the JUnit testing framework, other studies have investigated test smells in other frameworks. Baker *et al.* (2006) and Zeiss *et al.* (2006) identified test smells specific to Testing and Test Control Notation Version 3 (TTCN-3) test suites. Later, Counsell and Hierons (2007) explored the trade-offs of TTCN-3 refactorings. Results indicated that considering the dependencies among tests is essential to deciding whether to refactor a test code. Gatrell, Counsell and Hall (2009) investigated test code refactorings over 270 commercial C# software versions. Results indicated base refactorings are common, and complex structural refactorings are relatively rare. Bleser, Nucci and Roover (2019a) performed two empirical studies to analyze the diffusion of test smells at the class level of 164 open-source SCALA projects and assess the developers' perception of test smells. Results showed test smells have a low diffusion across test classes, and many developers perceived test smells but did not identify them.

Fernandes, Machado and Maciel (2021) analyzed the strategies for handling test smells in 90 open-source Python projects. As a result, the authors proposed and vali-

dated four test smells through a survey with 40 Python developers. Jorge, Machado and Andrade (2021) performed an empirical study with 11 open-source JavaScript projects. They aimed to investigate which test smells occur more frequently, whether they are likely to occur together, and whether the presence of test smells is related to classical bad design indicators on the test code. Aranega *et al.* (2021) analyzed Pharo, Java, and Python projects to investigate whether they exhibit similar categories of rotten green tests, i.e., a test method that always passes. As a result, the authors found all three languages contain smells related to conditional statements to stop the test method execution and assertion calls that force the test method to fail. Fushihara *et al.* (2023) analyzed test smells in Python and revealed test smells increase over commits and remain in test code as technical debt. Rwemalika *et al.* (2023) performed an exploratory analysis of test smells in system users' interactive tests with the Robot framework. Soares *et al.* (2023) analyzed manually tested software and contributed to a catalog and detection strategies for natural language test smells.

Besides having extensive ecosystems and tooling support available, Java and JUnit are mature technologies with well-established best practices and guidelines for software development and testing. Therefore, we contribute to the state-of-the-art with those technologies as they provide a solid foundation for conducting research on test smells, and are likely to have applicability to a large audience of software developers and researchers.

3.4 AUTOMATED TOOLS TO HANDLE TEST SMELLS

Aljedaani *et al.* (2021) listed 22 tools to detect test smells, from which only five tools support developers refactoring the test code. Most of those tools detect test smells in the test code written with the JUnit framework, whereas some focus on detecting specific test smells, such as redundant tests and dependency tests. Koochakzadeh and Garousi (2010b) released the TeReDetect (**Te**st **Re**dundancy **Detect**ion) tool that implements a strategy based on the code coverage to detect redundant tests. Later, Koochakzadeh and Garousi (2010a) improved on their tool (TeReDetect) and released the TeCReVis (**Te**st **C**overage and Test **Re**dundancy **Vis**ualization) plugin to visualize redundant tests and test coverage. To detect dependent tests, Zhang *et al.* (2014) released the DTDetector tool, Bell *et al.* (2015) released the ElectricTest tool, Biagiola *et al.* (2019) released the TEDD (Test Dependency Detector) tool, and Fraser, Gambi and Rojas (2020) released the PraDeT tool. Gyori *et al.* (2015) released the PolDet tool to detect dependencies regarding shared resources.

Other test smells can occur in test code written with the JUnit framework. Bavota *et al.* (2012) released an unnamed tool to detect the presence of nine test smells. Greiler *et al.* (2013) released the TestEvoHound tool to analyze the evolution of test smells related to the test fixture over the life of projects. Huo and Clause (2014) released the OraclePolish tool to detect brittle assertions and unused inputs. Palomba, Zaidman and Lucia (2018) released the Taste (Textual AnalySis for Test smEll detection) tool, which uses information retrieval techniques to detect test smells. Peruma *et al.* (2020a) released the tsDetect tool to detect 19 test smell. Virginio *et al.* (2020) extended the tsDetect tool and released the JNose Test tool, which detects 21 test smells and calculates code coverage. Later, Virginio *et al.* (2021) validated the JNose Test tool compared with the tsDetect tool.

In addition, Aljedaani *et al.* (2021) identified tools for detecting test smells in test code written with other testing frameworks. Reichhart, Gîrba and Ducasse (2007) released the TestLint tool to detect 27 test smells in SmallTalk test suites. Breugelmans and Rompaey (2008) released the TestQ tool to detect 12 test smells in C++ test suites. Bleser, Nucci and Roover (2019b) released the SoCRATES (SCala RAdar for TEst Smells) tool to detect the presence of six test smells in Scala. Delplanque *et al.* (2019) released the DrTest tool to detect rotten green tests in the Pharo ecosystem.

Concerning test code refactoring, Aljedaani *et al.* (2021) listed five refactoring recommendation tools to support developers in fixing test smells. Baker *et al.* (2006) released the TRex tool to analyze and refactor test smells specific to TTCN-3 test suites. Greiler, Deursen and Storey (2013) released the TestHound tool to detect six test smells related to test fixtures and recommend refactorings to fix them. Lambiase *et al.* (2020) released the DARTS (Detection And Refactoring of Test Smells) plugin, which utilizes information retrieval to detect test smells and rule-based refactorings to fix them. Santana *et al.* (2020) released the RAIDE plugin to detect two test smells and provide semi-automated support for refactoring the test code; after, Santana *et al.* (2022) and Santana *et al.* (2024) evaluated the usability of the RAIDE plugin. Martinez *et al.* (2020) released the RTj tool to detect and recommend test refactorings to fix rotten green test smells.

Complementary to the mapping performed by Aljedaani *et al.* (2021), we found other tools recently published. Marinke *et al.* (2019) developed the Neutrino plugin to detect and refactor seven test smells in JUnit test suites. Pecorelli *et al.* (2020b) extended the tsDetect tool to the VITRuM (VIsualization of Test-Related Metrics) plugin, which supports a visual interface of static and dynamic test-related metrics and

test smells. Wang *et al.* (2021) released the PyNose plugin to detect 17 test smells and Fernandes, Machado and Maciel (2022) released the TEMPY tool to detect ten test smells in Python built with the Unittest framework. Similarly, Bodea (2022) released the PyTest-Smell tool, which focuses on detecting test smells in Python built with the Pytest framework. Taniguchi, Matsumoto and Kusumoto (2021) introduced the JTDog plugin to detect three test smells in JUnit dynamically. Cruz and Costa (2020) released the TSVizzEvolution tool, which uses the output of the JNose Test tool to present the test smells evolution through a graphical interface. Paula and Bonifácio (2022) proposed TestAXE to support developers migrating for the JUnit5 framework and removing up to 13 test smells. Fulcini *et al.* (2022) proposed the FragilityLinter linter as a plugin with good practices to prevent test smells. Teixeira, Silveira and Guerra (2023) released the MeteoR plugin to detect test smells using the tsDetect tool (PERUMA *et al.*, 2020a), refactor them using Neutrino (MARINKE *et al.*, 2019), and identify the issues that can arise by checking the code coverage. Maier and Felderer (2023) released the SniffTest tool, which uses natural language processing to detect five test smells.

In order to support our research, we have extended some of the state-of-the-art tools to handle test smells. For example, we developed a tool that integrates the `tsDetect` (PERUMA *et al.*, 2020a), `VITRuM` (PECORELLI *et al.*, 2020b) and `TestRefactoringMiner` (MARTINS *et al.*, 2023a) to collect data on test smells, structural metrics, and test-specific refactorings from software repositories.

3.5 DEVELOPERS' PERCEPTION AND AWARENESS OF TEST SMELLS

While some studies explained the relationship between test smells and software quality, others pointed out developers sometimes perceive test smells as problematic. Rompaey, Bois and Demeyer (2006) proposed a metric-based heuristic approach to rank occurrences of test smells according to their relative significance. As a result, refactoring starting can be by the ranking. Tufano *et al.* (2016) surveyed 19 developers from open-source projects to investigate whether they could recognize occurrences of test smells in software projects. Besides, they performed an empirical study to investigate the survivability of test smells in 152 open-source projects belonging to two ecosystems (Apache and Eclipse). The results indicated that developers are unaware of test smells and hardly remove them from the test code. Spadini *et al.* (2020) argued developers could not recognize test smells as problematic due to the lack of thresholds. The authors analyzed 1,500 open-source

projects to identify thresholds for nine test smells and evaluated the thresholds with 31 developers from 47 open-source projects. Results included the definition of non-binary thresholds for four test smells, supporting the user-perceived maintainability impact.

Silva Junior *et al.* (2020) and Silva Junior *et al.* (2021) surveyed 60 practitioners to investigate their awareness of test smells. Results indicated practitioners introduce test smells during their daily programming tasks. However, the practitioners' experience cannot be considered a root cause for the insertion of test smells in the test code. Santana *et al.* (2021) surveyed 87 practitioners and interviewed eight other practitioners to investigate their perception of test smells and strategies to handle them. Results indicated most participants consider they should refactor test smells but do not always do it. Campos, Rocha and Machado (2021) interviewed six developers to investigate their perception of the severity of test smells in their developed test code. Results indicated despite developers perceiving test smells as non-severe, they can negatively impact the project. Chen, Embury and Vigo (2023) surveyed developers to understand whether they consider test smells as sources of technical debt and whether removing them is worth spending effort. Results showed some test smells rarely occur and do not show a consistent pattern of quick or delayed removal.

Other studies investigated how the developers' expertise can influence their perception of test smells. In particular, Lima *et al.* (2023) investigated whether the developers' profiles and experience can influence their perception of ten different test smells. The developers exhibited a low level of agreement, mostly influenced by specific heuristics applied to detect test smells. Campos *et al.* (2023) and Campos, Martins and Machado (2023) conducted two empirical studies to investigate the relationship between developers' experience and the survivability of test smells during test code refactoring. As a result, they found that 67.28% of test smells are inserted during test class creation, while 20.88% of test smells are removed during project evolution. Additionally, core developers are responsible for inserting 88.91% of test smells and removing 89.82% of test smells. Damasceno *et al.* (2023a) and Damasceno *et al.* (2023b) surveyed 20 developers while removing five types of test smells to investigate the impact of test smell refactoring on quality attributes, the influence of developers' experience, and their perceptions of refactoring test smells. Results showed an improvement in quality attributes, such as cohesion and complexity. Additionally, less experienced developers took more time to refactor test smells compared to experienced ones.

Concerning the awareness of test smells, the studies discussed above focused on the

existing catalog of test smells. This approach can lead to a limited perspective, potentially causing developers to overlook or misjudge certain test code issues not explicitly listed in the catalog. In contrast, we did not rely on existing catalogs of test smells and refactorings but identified the smells that developers find problematic in practice. Additionally, the previous studies sought developers' opinions on test smells, which vary based on their experiences, viewpoints, and familiarity with certain code practices. Instead, we mined refactorings from the projects' commit history to reveal developers' actual decisions regarding addressing test smells. This approach potentially minimizes the subjectivity in developers' awareness and perceptions of test smells (RQ_1).

3.6 REFACTORINGS TO FIX TEST SMELLS

Peruma *et al.* (2020b), have investigated the relationship between refactoring changes and their effect on test smells. The authors used the Refactoring Miner tool to detect refactoring operations and the tsDetect tool to identify the test smells from unit test files of 250 open-source Android Apps. Results showed refactoring operations in test and non-test files differ, and refactorings co-occur with test smells. However, refactorings occur for reasons other than fixing test smells. Similarly, Kim, Chen and Yang (2021) conducted an empirical study on the evolution and maintenance of test smells in 12 open-source projects. The authors analyzed the commits that removed test smells and concluded the removal of test smells was due to maintenance activities.

A second line of research is represented by qualitative studies targeting the developer's perception of test refactoring. Soares *et al.* (2020) investigated how developers refactor test code to eliminate test smells. The authors surveyed 73 open-source developers to assess their preference and motivation to choose between 10 smelly and refactored test code samples. Next, they submitted 50 pull requests to assess developers' acceptance of the proposed refactorings. The results showed developers preferred the refactored test code for most test smells. In another work, Soares *et al.* (2022) investigated whether the JUnit5 features help refactor test code to remove test smells. They conducted a mixed-method study to analyze the usage of JUnit5 features in 485 popular Java open-source projects. They identified the features helpful for test smell removal and proposed novel refactorings to fix test smells. The results indicated that using the JUnit5 features was only 17.6% during the test code creation and maintenance.

Pizzini, Reinehr and Malucelli (2023a) proposed and evaluated a method to address

the ET test smell through experiments, comparing the original version of the test code with its refactored version. Results indicated an increase in test execution time and Lines of Code (LOC) while removing the test smell, with no occurrence of test errors or failures. Pizzini, Reinehr and Malucelli (2023b) investigated the effects of automatic refactoring on test codes. The authors concluded that the effects are related to compilation errors, execution failures, and changes in the behavior of unit tests. Additionally, they presented a process for developing automatic refactoring tools aimed at improving test code quality by eliminating test smells.

When asking developers to choose between smelly and refactored test code, a notable observation was their lack of awareness of features provided by testing frameworks. This knowledge gap could lead to the rejection of certain refactoring strategies intended to fix test smells. In contrast, we analyzed the actual refactorings developers performed in test code over time, providing a more objective insight into how they tackle test smells. Understanding these developer practices not only enlightens us on the prevalent features regularly employed in test code but also empowers us to propose test refactorings that better align with developers' perspectives and preferences (RQ_1).

3.7 MACHINE LEARNING TECHNIQUES TO HANDLE TEST SMELLS

Given the gap between developers' perceptions and approaches implemented by tools for detecting test smells and refactoring the test code, other studies have investigated the feasibility of using ML techniques to handle test smells. Martins *et al.* (2021a) used structural metrics of test code to train ML algorithms for classifying four test smells. Results indicated the algorithms performed well in detecting test smells, especially the Random Forest algorithm. Similarly, Hadj-Kacem and Bouassida (2021) analyzed the agreement level among the detection tools, and they suggested a multi-label classification approach to detect test smells based on a deep representation of the test code. Similarly, they found the Random Forest algorithm presented the best results, alleviating the limitations of heuristic-based techniques.

While some studies have used ML techniques to detect test smells, we observe a lack of studies that explicitly consider the classification of test code refactoring operations. The closest work is the one by Aniche *et al.* (2022), in which the authors proposed an approach to predict refactoring operations only in production code using process and structural metrics as predictors. Our study addresses this gap by focusing on test

code, considering its peculiarities and the evolutionary nature of the test source code—a perspective not explicitly considered by Aniche *et al.* (2022) (RQ_3).

3.8 LIMITATIONS OF PRIOR WORK

As some limitations of prior work, we can cite:

- The lack of agreement among the detection tools regarding the definition of test smells and accuracy leads developers not to perceive test smells as problematic. In addition, most of the test smells defined in the literature focus on the JUnit4 features. Once the testing framework has evolved, many test smells can no longer represent problems in the test code. It suggests that we should focus on the developers' practices to derive more appropriate metrics or approaches for suggesting more assertive information to developers;
- Studies investigating the correlation of test smells and structural metrics aimed to highlight a potential relationship between the production code and test code. Therefore, there are still gaps in expanding state-of-the-art practices by providing a deeper understanding of test smells and their relationship with the test quality;
- Few studies investigated the relationship of test smells with coverage, faults, and flakiness. It would be interesting to replicate or apply alternative experimental settings to understand how well the previous findings generalize;
- Although many studies investigate developers' perceptions of test smells, few consider the extent to which developers' experience and roles during software development influence test code quality. Deeper investigations into this area could explore varying levels of experience, different roles within development teams, and overall familiarity with testing best practices that contribute to handling test smells. Such research could provide valuable insights into how developers' backgrounds and expertise impact their ability to address test smells and enhance test code quality effectively;
- Although the studies provide replication packages, most have become inaccessible. The datasets contain information from the different sets of projects regarding the presence and location of test smells and other structural metrics; such datasets do not store the test codes. The assessment of the detection tools uses

datasets containing different information, making it hard to compare their accuracy. Therefore, more effort is necessary to build Findable, Accessible, Interoperable, and Reusable (FAIR) datasets (KATZ; GRUENPETER; HONEYMAN, 2021) with metrics calculated from the test code and the test code itself;

- There are numerous tools available for detecting test smells, primarily based on heuristics and rules. However, only a few of these tools support refactoring the test code to address test smells. Exploring the potential of an automated test refactoring recommendation tool guided by a developer-centric approach holds substantial promise. Such a tool could identify refactoring candidates and suggest appropriate refactorings to improve test code quality. This approach could significantly assist developers in efficiently addressing test smells and enhancing the overall quality of their test suites.
- Studies using ML techniques require executing tools on the production or test code to calculate metrics to serve as features for the ML algorithms. Other ML techniques using pre-trained language models can rely solely on test code to detect whether a test is smelly or not

In comparison to the current literature on test smells, we make several contributions to the scientific community through four main investigations. Firstly, we delve into developers' practices in open-source projects to derive a catalog of test-specific refactorings aimed at addressing test smells. Secondly, we employ supervised machine learning (ML) algorithms to classify developers' intentions in applying test refactoring and specific test refactoring operations. This marks a significant step towards the development of an automated test refactoring recommendation tool guided by a developer-centric approach. Thirdly, we analyze whether test refactorings are driven by low-quality test codes and to what extent these refactorings contribute to improving code quality. To support these investigations, we also contribute to the community by releasing datasets and extensions of state-of-the-art tools for repository mining. These tools enable the collection of structural and process metrics, test smells, and test refactorings, facilitating further research in this area.

3.9 CHAPTER SUMMARY

This chapter reported on the results of an ad-hoc review on test smells. It mapped 95 studies in the literature to understand the approaches and tools to handle test smells.

Despite an increasing interest in test smells, the review led us to claim the need for more effective metrics, methods, and techniques for detecting test smells and test-specific refactorings to fix them.

Given the lack of agreement among the tools to handle test smells aligned with the advances of the testing frameworks, developers cannot recognize some test smells as a problem of the current development practices in the test code. Consequently, there emerges a demand for practical, real-world guidelines that enable developers to recognize the detrimental impact of test smells.

In response, ML techniques become a promising attempt to detect test smells and suggest refactorings closer to the current development practices. Thus, shifting towards ML addresses the current gaps in tooling and holds great potential to enhance the effectiveness and adaptability of test smell detection and refactoring to different scenarios.

Chapter 4

MINING TEST REFACTORINGS IN PRACTICE

To keep up with the advances in testing frameworks or improve the test code structure by eliminating test smells, developers should refactor the test code in a way that does not change test logic (FOWLER, 1999). Although refactoring has been extensively studied in the literature to address code smells (VIDAL; MARCOS; DÍAZ-PACE, 2016; CEDRIM *et al.*, 2017; TUFANO *et al.*, 2017b; PANTIUCHINA *et al.*, 2020; LACERDA *et al.*, 2020), test code can exhibit unique smells that reflect issues with its organization, implementation, or even interaction with other test code, requiring a different set of refactorings (DEURSEN *et al.*, 2001). According to recent mapping literature (ALJEDAANI *et al.*, 2021), recent studies have proposed strategies and tools for detecting and refactoring test smells. However, existing tools are not widely used in practice as they implement simplistic strategies based on predefined thresholds or rules to detect test smells (SPADINI *et al.*, 2020; PANICHELLA *et al.*, 2022). In addition, those tools implement refactoring strategies not derived from current development practices, and they lack support for many problems developers face when refactoring the test code (PANICHELLA *et al.*, 2022; SOARES *et al.*, 2022; PERUMA *et al.*, 2022).

Considering the gap between the research on test code refactoring and its adoption in practice, we argue analyzing the refactorings performed by developers may provide insights into the problems they commonly face and the current development practices they use to refactor the test code. This chapter presents an empirical study to answer **RQ$_1$: How do developers perform test code refactorings to fix test smells in open-source projects?**. First, we analyzed the change history of 13 open-source JAVA

projects with test cases written with either JUnit4, JUnit5, or both over three years to catalog the refactoring operations performed in the test code to address test smells. Second, we sought feedback from software developers who have contributed to the projects included in our dataset to allow gathering their perspectives on several aspects regarding test smells and refactorings.

The remainder of this chapter is structured as follows. Section 4.1 presents our goals and research questions. Section 4.2 presents the data collection and analysis approach to catalog test refactorings performed in practice. Section 4.3 reports and describes the refactoring operations found in practice for test smells. Section 4.4 discusses the practical implications of test-specific refactorings. Section 4.5 points out the possible threats that could affect our results.

4.1 RESEARCH QUESTIONS AND OBJECTIVES

The *goal* of our empirical study is to analyze the test refactoring operations performed by developers in practice, with the *purpose* of creating a catalog of the test smells developers deem as problematic and which test refactoring operations developers apply to fix the test smells. The *perspective* is both researchers and practitioners who are interested in the problems affecting the test code and how to solve them.

More specifically, we aim to answer the research question (**RQ**):

> $\mathbf{RQ}_{1.1}$. *What common refactoring operations do developers apply to fix test smells in the test code?*

Through $\mathbf{RQ}_{1.1}$, we aim to manually analyze the changes applied to a version v_n of a test class to classify the refactoring operations applied by developers and its previous version v_{n-1} to classify the test smell fixed through the refactoring operation. Once we have the pairs of smelly and refactored test code, we can generate a catalog to specify the test-specific refactorings for fixing test smells.

Upon completion of this investigation, we further elaborate on the acceptance of our catalog, addressing the **RQ**:

> $\mathbf{RQ}_{1.2}$. *To what extent are refactoring operations relevant to developers handling test smells?*

Through $\mathbf{RQ}_{1.2}$, we investigate the developers' perception of the usefulness of our

catalog in practice. In addition, we perform a comparison of our catalog with the existing ones in the current literature to highlight the novel refactoring operations found in our investigation.

4.2 APPROACH TO DERIVE A CATALOG OF TEST REFACTORINGS

In order to answer $\mathbf{RQ}_{1.1}$, we follow the approach in Figure 4.1 to analyze the test code changes. The approach consists of four main steps: *Selecting subject systems* (Section 4.2.1), *Extracting test code changes* (Section 4.2.2), *Classifying test code changes* (Section 4.2.3), and *Deriving a catalog of test refactorings* (Section 4.3). We made all artifacts generated through our approach publicly available (MARTINS *et al.*, 2023b).

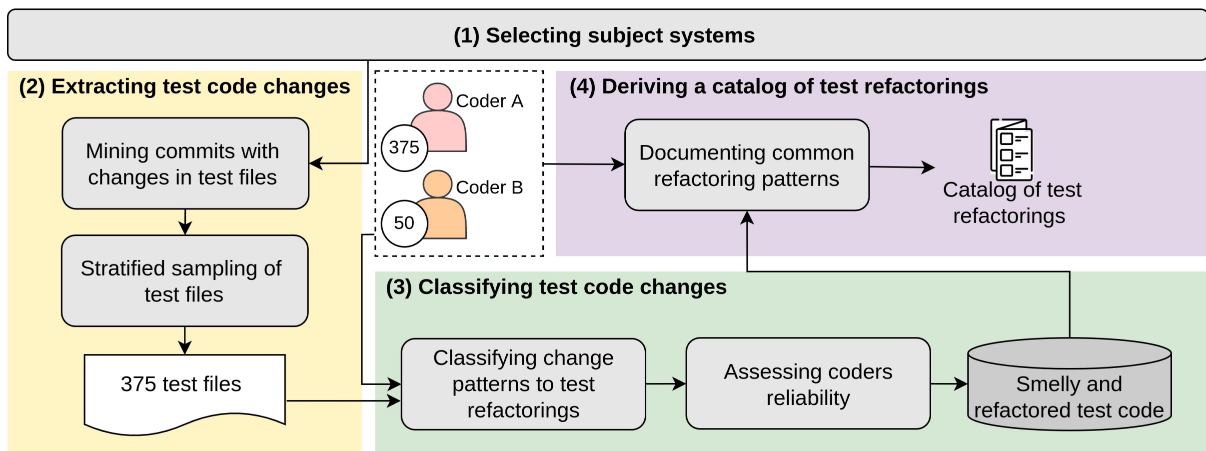


Figure 4.1: Overview of the Proposed Approach.

4.2.1 Selecting subject systems

We analyzed 13 open-source JAVA projects from Apache Foundation selected by Kim, Chen and Yang (2021), covering different domains, from big data processing and warehousing solutions to distributed databases and programming languages. Test suites of those projects are written using either JUNIT4, JUNIT5, or sometimes a combination of both. Some of those projects migrated from JUNIT4 to JUNIT5.

4.2.2 Extracting test code changes

To extract the test code changes, we automatically analyzed the commit history of the projects, and we selected a stratified random sample of modified test files from each project for manual analysis through the following activities.

Mining commits with changes in test files. For each project, we analyzed all commits over one three-year period for two main reasons. First, analyzing modifications in project revisions might hide test code refactorings since revisions often exhibit changes in software functionality (CEDRIM *et al.*, 2017). Second, diving into modifications at the commit level offers insights into minor improvements in the test code (CEDRIM *et al.*, 2017) but at the expense of the computational time required for mining all test code modifications and the manual effort required to analyze them (TUFANO *et al.*, 2017a). Therefore, considering the latest advancements in the JUnit testing framework, we restricted our analysis to the commit level between 2019 and 2021 to understand how developers refactor test code to address test smells.

We developed a Git commit history analyzer using the JGit API[1] to extract all the commits of the set of projects between 2019 and 2021. The JGit API is a Java library that implements all the Git commands. Given a set of files as input, our analyzer selects the commits related to changes in the test files and discards the other commits from further analysis. A commit changes the test file if the involved files have the extension `.java` and a prefix or a suffix of `[Tt]est(s*)` (at least in the current commit, to indicate a rename of a test file). Lastly, the analyzer stores the information obtained from each test file in the commits. As a result, the analyzer identified 18,532 commits with 132,819 modified test files (see Table 4.1).

Stratified sampling of test files. We selected a statistically significant sample of 375 modified test files. First, we analyzed whether the commit message suggests changes in the test files or whether the changes represent the co-evolution between test and production code. Therefore, we developed a script to parse all the commit messages of the 18,532 commits and only select the relevant ones containing the "test" word and, optionally, the "improvement" or "refactoring" words. As a result, we identified 3,786 relevant commits containing 14,829 modified test files with potential test-specific refactorings. Second, we

[1] Available at: <https://www.eclipse.org/jgit/>

applied stratified random sampling with a 95% confidence level and a 5% confidence interval. Hence, given that our variable of interest is the potential refactorings in test files (Table 4.1: *Test files w/ potential refactorings*), we randomly sampled a certain number of test files from each project relative to its proportion in the dataset, ensuring that each project is sufficiently represented in the sample. As a result, we selected 375 test files from all projects with potential refactorings. Table 4.1 shows the distribution of test files per project.

Table 4.1: Characterization of the studied projects in terms of the number of commits and selected test files

#	Projects	Commits w/ modified test files	Modified test files	Commits w/ potential refactorings	Test files w/ potential refactorings	Sample of test files w/ potential refactorings
1	Accumulo	315	1,691	129	820	21
2	Bookkeeper	198	625	61	362	10
3	Camel	5,295	80,219	1,200	7,133	180
4	Cassandra	650	2,143	200	483	12
5	Cxf	691	10,230	395	961	24
6	Flink	7,152	25,451	237	490	12
7	Groovy	187	502	29	47	1
8	Hadoop	183	375	89	109	3
9	Hive	101	135	23	28	1
10	Kafka	2,896	8,751	1,141	3,463	88
11	Karaf	209	484	57	198	5
12	Wicket	294	908	58	133	3
13	Zookeeper	361	1,305	167	602	15
	Total	18,532	132,819	3,786	14,829	375

4.2.3 Classifying test code changes

We conducted a manual analysis on a statistically significant sample of 375 modified test files to classify the patterns of changes related to test smells and refactoring operations. Our activities included:

Classifying change patterns to test refactorings. We manually analyzed the ***`git diff`*** representing the changes in the test files. Two coders (persons) applied open coding to label test smells (in parent commits) and refactoring operations (in current commits). In particular, we analyzed whether the removed lines of code in the parent commit are related to test smells (e.g., assertion parameters, `try/catch` blocks, conditional or loop

structures, threads, print statements, annotation tags, and methods signature). Then, we analyzed whether the added lines of code address test smells without changing the test logic. It is worth noting changes in test files can add or remove test classes or methods, but we only consider refactoring whether they address a certain smell in the test code. We standardized all the labels assigned by coders by cross-referencing them with the existing literature.

Listing 4.1 presents a unified ***git diff*** for the `testHelloWorldWithDummyPlugin` test method in the `CodeGenTest` class from the `Cxf` project (APACHE CXF, 2019b). In that figure, (i) two line numbers precede each line to indicate its relative location in the parent and current commit, respectively, (ii) a '-' or '+' symbol precedes each modified line where the '-' symbol indicates removals in the parent commit (highlighted lines in red) and the '+' symbol indicates additions in the current commit (highlighted lines in green), (iii) highlighted lines in blue indicate omitted lines and (iv) a header indicates the test method in which the change occurred. In some cases, labels were inconsistent across coders, though they referred to the same test smell. For example, when analyzing the modifications in the `testHelloWorldWithDummyPlugin` test method, Coder A labeled the parent commit (line 644 highlighted in red) with a *Simplify logic* test smell and Coder B with an Inappropriate Assertion (InA) test smell; both coders labeled the current commit (line 644 highlighted in green) with a *Modify Assert Type* refactoring. To resolve labeling inconsistency, we searched for the test smells and refactoring operations in the literature to standardize the labels of the two coders. As a result, in the above example, we agreed to use the InA (KUMMER; NIERSTRASZ; LUNGU, 2015) and the *Modify Assert Type* refactoring instead of *Simplify logic* test smell.

```
...        @@ -642,7 +642,7 @@ public void testHelloWorldWithDummyPlugin() throws Exception {
642  642       Class<?> clz = classLoader.loadClass("org.apache.cxf.w2j.hello_world_soap_http.types.SayHi");
643  643       Method method = clz.getMethod("dummy", new Class[] {});
644      -     assertTrue("method declared on SayHi", method.getDeclaringClass().equals(clz));
     644 +     assertEquals("method declared on SayHi", method.getDeclaringClass(), clz);
645  645    }
...
```

Listing 4.1: The ***git diff*** of the `CodeGenTest` class of the `Cxf` project (APACHE CXF, 2019b). The diff highlights in red the lines removed, and in green the lines added.

Assessing coders' reliability. To assess the reliability of the manual classification, two researchers (Coder A and Coder B) analyzed a set of test files to calculate the Kappa

statistics (COHEN, 1960). Coder A analyzed the 375 modified test files and classified the diff in the test code changes regarding the test smells and the refactoring operations. For example, Coder A analyzed the instance presented in Listing 4.1 and classified the lines highlighted in red as an InA test smell (diff - left side) and the lines highlighted in green as a *Modify assert type* refactoring (diff - right side). Then, we randomly sampled 50 modified test files out of 375 for Coder B to classify the test smells and refactorings. Coder A and Coder B found 198 instances containing pairs of smelly and refactored test codes in the same set of test files. The agreement level between the coders was high; they agreed on 196 instances, and each one missed two instances (Cohen's kappa = 0.98). Next, a third researcher (Coder C) joined the discussion to classify the four missed instances. The coders added the four missed instances in the final set, totaling 200 instances. The final set contains 611 smelly and refactored test code pairs from 156 test files (i.e., 41.7% of the modified test classes are related to test smells); 200 pairs identified by Coder A and Coder B, plus 413 (611 - 198) pairs identified by Coder A. In addition, Coder C helped standardize the labels given to each instance. For example, while Coder A classified some instances of test cases with no assertion as the Unknown Test (UT) test smell, Coder B recognized a problem in the test code but did not know the test smell name, classifying the instances with *No test smell*. During the discussions, we agreed on classifying those instances as the UT test smell.

4.3 DERIVING A CATALOG OF TEST-SPECIFIC REFACTORINGS

After classifying test code modifications, we presented the TSR-Catalog, a catalog for Test Smells Refactorings (MARTINS *et al.*, 2023c), outlining the reengineering process for conducting test-specific refactorings and leveraging the identified patterns. That catalog was the foundation for deriving systematically organized insights into how developers refactor test code in practice, and then, we compared it with existing literature.

Although the literature reports more than 180 test smells (GAROUSI; KüçüK, 2018), we found nine test smells in our stratified sample of refactored test smells. Table 4.2 summarizes the detection and test-specific refactorings of the test smells found (highlighted in gray). The *Detection* and *(%)* (1st and 2nd) columns present 13 rules for detecting the test smells and their respective distributions in the dataset, e.g., *(2) Identify* ***`@Test expected`*** *annotation* corresponds to 80.4% instances of Exception Handling (ECT) test smell. The *Refactoring* and *(%)* (3rd and 4th) columns present 11 refactoring operations to solve test smells and their respective frequency in the dataset, e.g., *(b) Replace*

Table 4.2: Summary of detection and refactorings for test smells

Detection	(%)	Refactoring	(%)	
Exception Handling (336 instances)				
(1) Identify `try/catch` blocks	14.6	(a) Replace `try/catch` block with `@Test expected` annotation	0.0	☆
(2) Identify `@Test expected` annotation	80.4	(b) Replace `try/catch` block, `@Test expected` or `@Rule` annotations with the `assertThrows`	100.0	★
(3) Identify `@Rule` annotation	5.0			
Inappropriate Assertion (40 instances)				
(4) Identify the not operator within a parameter	22.5	(c) Replace the not operator within the assertions	22.5	✪
(5) Identify conditional expressions as parameter	25.0	(d) Split conditional expressions into two different parameters	25.0	✪
(6) Identify reserved words as parameters	52.5	(e) Replace reserved words with inappropriate assertion	52.5	✪
Assertion Roulette (135 instances)				
(7) Identify undocumented assertions in a test method	100.0	(f) Add an explanation message to the assertion	100.0	★
		(g) Split assertions into single test methods	0.0	☆
		(h) Surround assertions with assertAll	0.0	☆
Bad Naming (30 instances)				
8) Identify test classes' names without the word "Test" in it	100.0	(i) Rename test classes	100.0	★
Ignored Test (2 instances)				
(9) Identifying ignored methods	100.0	(j) Code Removal	100.0	★
Redundant Print (2 instances)				
(10) Identifying calls for the `System.out.print` method	100.0	(j) Code Removal	100.0	★
Empty Test (4 instances)				
(11) Identifying methods with no executable body	100.0	(j) Code Removal	50.0	★
		(k) Code Addition	50.0	★
Unknown Test (50 instances)				
(12) Identifying methods with an executable body but no assertions	100.0	(k) Code Addition	100.0	★
Sleepy Test (12 instances)				
(13) Identifying Thread.sleep method	100.0	(k) Code Addition	100.0	★

Classification: (★) refactorings from literature applied in practice, (✪) applied in practice but not found in literature (☆), and from literature not applied in practice

`try/catch` *block,* `@Test expected` *or* `@Rule` *annotations with the* `assertThrows` fixed 100% instances of the ECT test smell. Finally, the classification symbols show whether the test-specific refactorings are i) novel or proposed in the literature and ii) applied in practice.

We have the following distribution when considering the pairs of test smells and refactorings in the entire dataset. The ECT test smell consists of 49 instances for the 1-b pair (8.0%), 270 instances for the 2-b pair (44.2%), and 17 instances for pairs 3-b (2.8%) (Section 4.3.1). The Inappropriate Assertion (InA) test smell consists of 9 instances for the 4-c pair (1.5%), 10 instances for the 5-d pair (1.6%), and 21 instances for the 6-e pair (3.5%) (Section 4.3.2). The Assertion Roulette (AR) test smell consists of 135 instances for the 7-f pair (22.1%) (Section 4.3.3). The Bad Naming (BaN) test smell consists of 30 instances for the 8-i pair (4.9%) (Section 4.3.4). The Ignored Test (IgT) test smell consists of 2 instances for the 9-j pair (0.3%). The Redundant Print (RP) test smell consists of 2 instances for the 9-j pair (0.3%). The Empty Test (EpT) test smell consists of 2 instances for the 11-j pair (0.3%) and 2 instances for the 11-k pair (0.3%). The Unknown Test (UT) test smell consists of 50 instances for the 12-k pair (8.2%). Finally, the Sleepy Test (ST) test smell consists of 12 instances for the 13-k pair (2.0%) (Section 4.3.5).

The last five test smells in Table 4.2 were addressed using irregular refactoring operations (i.e., different from one instance to another) performed through miscellaneous code additions or deletions in the current commit. This resulted in a lack of identifiable patterns to precisely describe their refactoring, making them not aligned with the refactoring techniques outlined in our catalog. While the refactorings involving only code deletion were trivial, code addition requires a deeper understanding of the production code to develop the test code. Therefore, we did not extensively describe these test smells and their refactorings as others, but we present some examples from our dataset in Section 4.3.5.

4.3.1 The Exception Handling test smell

Meszaros (2007) defined the Exception Handling (ECT) test smell to verify whether the error scenarios have been coded correctly by using the language constructor to catch errors. Peruma *et al.* (2019) stated the ECT test smell occurs when developers write custom exception-handling code or throw an exception. Instead, developers should use

the exception handling of JUnit to automatically pass or fail the test method.

In practice, developers use three different test structures that lead to the insertion of the ECT test smell, and they can use two refactoring strategies defined in the literature to fix it. Figure 4.2 illustrates the detection and refactoring operations to fix the ECT test smell. The circles indicate the test smell detection, whereas the squares represent the refactoring operations. In that figure, $stmt$, $stmt'$, and $stmt''$ are a set of statements, $stmt'_i \in stmt'$ statements that raise the E exceptions, evs is an optional exception verification, and M is an optional message.

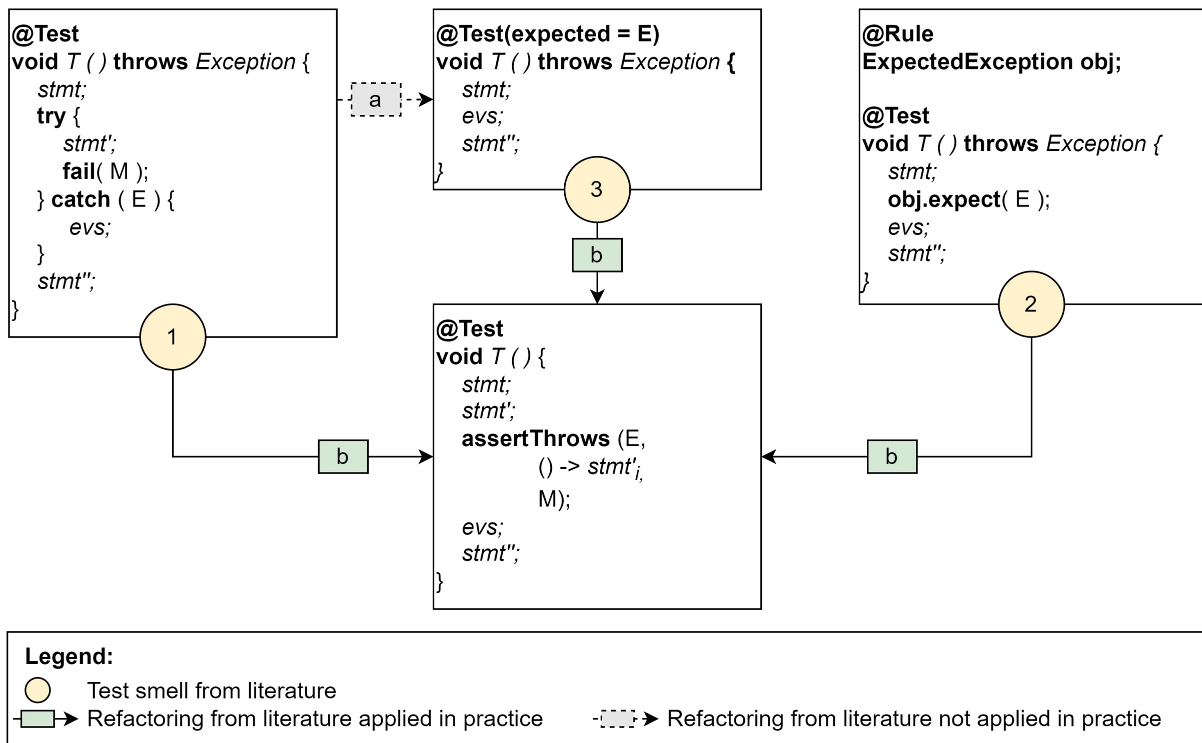


Figure 4.2: Detection and refactoring the ECT test smell.

Detecting the Exception Handling test smell

(1) ***Identify `try/catch` blocks.*** Peruma *et al.* (2019) state the ECT test smell occurs when a test method contains either a `throw` statement or `catch` statement. Following that definition, the JNose Test and tsDetect tools identify a `try/catch` block in the test method to detect the ECT test smell.

(2) ***Identify `@Rule` annotation.*** The `@Rule` annotation checks whether a test method throws an exception through the `ExpectedException` rule. Unlike the `try/catch` block, the `@Rule` annotation does not require the developer to customize the exception handling

or throw an exception to pass or fail the test. It means the test method passes if the statements within it did not result in an exception. Although previous studies (KIM; CHEN; YANG, 2021; SOARES *et al.*, 2020) showed developers are unaware of the `@Rule` annotation, we found developers using and fixing it in practice.

(3) *Identify `@Test expected` annotation.* The `@Text expected` annotation indicates that an exception can be thrown anywhere in the test method. The annotation makes it hard for developers to track which statement fails the test (KIM; CHEN; YANG, 2021; SOARES *et al.*, 2020).

Refactoring the Exception Handling test smell

(a) *Replace the `try/catch` block with the `@Test expected` annotation.* Meszaros (2007) and Peruma *et al.* (2019) suggested using JUnit features instead of the `try/catch` blocks. JUnit4 provides the expected attribute of the `@Test expected` annotation and the `Rule` check to handle exceptions. According to Peruma *et al.* (2019), developers should split the test method into multiple test methods to verify different error scenarios. Besides, test methods that generate exceptions should contain the `@Test expected` annotation to fail when the exception occurs. The refactoring consists of the following operations:

1. Add an `@Test expected` annotation as a parameter in the E exception class. The exception class is a parameter in the `catch` statement.
2. Copy the code within the `try/catch` block containing the *stmt* statements, the $stmt'$ statements that raise the exception, the *evs* optional exception verification, and the $stmt''$ optional statements.
3. Remove the `try/catch` block.

We did not find developers performing that refactoring in practice. But, they perceive the refactoring suggested by Peruma *et al.* (2019) as an improvement.

(b) *Replace the `try/catch` block, `@Test expected`, and `@Rule` annotations with the `assertThrows` method.* More recently, Soares *et al.* (2022) investigated whether the developers use the `assertThrows` method of JUnit5 to handle exceptions. In practice, we identified developers using the `assertThrows` method for refactoring the `try/catch` block and the `@Test expected` and `@Rule` annotations. The refactoring consists of the following operations:

1. Add an `assertThrows` method:

- Pass the E exception class as the first parameter of the `assertThrows` method. The exception class is a parameter in the `catch` statement or within the `@Test expected` annotation.
- Pass the $stmt'_i$ statements from the set of $stmt'$ statements that raise the E expected exception within a lambda expression passed as a parameter of the `assertThrows` method. Instead of identifying the $stmt'_i$ statements that raise an exception, the developers can optionally select all statements $stmt$, $stmt'$, $stmt''$, and the *evs* exception verification to compose a lambda expression.
- Pass the M fail message as the optional parameter in the `assertThrows` method to describe the assertion.

2. Copy the first *stmt* statements before the `assertThrows` method.
3. Copy the *evs* optional exception verification and the further $stmt''$ optional statements after the `assertThrows` method.
4. Remove the `try/catch` block, the `@Rule` annotation, or the `@Test expected` annotation.
5. Remove the `throws` exception from the method signature.

Samples from practice

Replacing the `try/catch` block with the `assertThrows` method (1-b). Listing 4.2 presents a `try/catch` block within the `testBadConfiguration` test method of the `Camel` project (APACHE CAMEL, 2021b). The refactoring consists of removing the `try/catch` block (lines 109 - 114, highlighted in red) and adding the parameters in the `assertThrows` method (lines 113 - 114, highlighted in green): i) the `ResolveEndpointFailedException` exception class found in the `catch` statement, ii) the call for the `sendBody` production method raises the exception, and iii) the optional message explains the assertion.

Replacing the `@Rule` annotation with the `assertThrows` method (2-b). Listing 4.3 presents the `UnsynchronizedBufferTest` test class of the `Accumulo` project, declaring the `ExpectedException` rule through the `@Rule` annotation and checking whether the exception behaves as expected in the `testByteBufferConstructor` method (APACHE ACCUMULO, 2021). The refactoring consists of removing the `@Rule` annotation (lines 35 and 36, highlighted in red) of the class and the `expected exception` from the method (lines 57 - 59, highlighted in red). Then, the refactoring consists of adding the `assertThrows` method in the test method with the parameters: i) the optional message that explains the assertion, ii) the `ArrayIndexOutOfBoundsException` exception

```
107 108   @Test
108     - public void testBadConfiguration() throws Exception {
109     -   try {
110     -     template.sendBody(String.format("mina:tcp://localhost:%1$s?sync=true&codec=#XXX", getPort()), "Hello World");
111     -     fail("Should have thrown a ResolveEndpointFailedException");
112     -   } catch (ResolveEndpointFailedException e) {
113     -     // ok
114     -   }
    109 + public void testBadConfiguration() {
    110 +   final int port = getPort();
    111 +   final String format = String.format("mina:tcp://localhost:%1$s?sync=true&codec=#XXX", port);
    112 +
    113 +   assertThrows(ResolveEndpointFailedException.class, () -> template.sendBody(format, "Hello World"),
    114 +     "Should have thrown a ResolveEndpointFailedException");
```

Listing 4.2: Diff between the original and refactored `testBadConfiguration` method of the `MinaCustomCodecTest` class of the `Camel` project (APACHE CAMEL, 2021b).

```
35      - @Rule
36      - public ExpectedException thrown = ExpectedException.none();
37      -
38   34   @Test
39   35   public void testByteBufferConstructor() {
40   36     byte[] test = "0123456789".getBytes(UTF_8);
...       @@ -54,9 +50,11 @@ public void testByteBufferConstructor() {
54   50     assertEquals("34567", new String(buf, UTF_8));
55   51
56   52     buf = new byte[6];
57      -   // the byte buffer has the extra byte, but should not be able to read it...
58      -   thrown.expect(ArrayIndexOutOfBoundsException.class);
59      -   ub.readBytes(buf);
     53 +
     54 +   final UnsynchronizedBuffer.Reader finalUb = ub;
     55 +   final byte[] finalBuf = buf;
     56 +   assertThrows("the byte buffer has the extra byte, but should not be able to read it",
     57 +     ArrayIndexOutOfBoundsException.class, () -> finalUb.readBytes(finalBuf));
60   58   }
```

Listing 4.3: Diff between the original and refactored `testByteBufferConstructor` method of the `UnsynchronizedBufferTest` class of the `Accumulo` project (APACHE ACCUMULO, 2021).

class found in the `thrown.expect` method, and iii) the call for the `readBytes` production method raising the exception (lines 54 - 58, highlighted in green).

Replacing the `@Test expected` annotation with the `assertThrows` method (3-b). Listing 4.4 presents the `testNonDefaultConfig` test method of the `Camel` project (APACHE CAMEL, 2019a), which uses the `@Test expected` annotation to handle the exception (line 41, highlighted in red). The refactoring consists of removing the `@Test`

`expected` annotation and adding the `assertThrows` method with the parameters: i) the `IllegalArgumentException` exception class found in the `@Test expected` annotation, and ii) the statements raising or not the exception (44 - 51, highlighted in green). It is worth noting that developers consider the `@Test expected` annotation hard to trace the method raising the exception, which justifies inserting all the statements in the lambda expression.

```
41      -  @Test(expected = IllegalArgumentException.class)
42      -  public void testNonDefaultConfig() throws Exception {
43      -     TelegramComponent component = (TelegramComponent) context().getComponent("telegram");
44      -      component.setAuthorizationToken(null);
45      -     component.createEndpoint("telegram:bots");
     44 +  @Test
     45 +  public void testNonDefaultConfig() {
     46 +     assertThrows(IllegalArgumentException.class, () -> {
     47 +        TelegramComponent component = (TelegramComponent)context().getComponent("telegram");
     48 +        component.setAuthorizationToken(null);
     49 +        component.createEndpoint("telegram:bots");
     50 +     });
     51     }
```

Listing 4.4: Diff between the smelly and refactored `testNonDefaultConfig` of the `TelegramComponentParametersTest` class of the `Camel` project (APACHE CAMEL, 2019a).

4.3.2 The Inappropriate Assertion test smell

The `Assert` class has `assert` methods that differ in parameters and semantics about what they assert. For example, the `assertTrue` method receives a mandatory parameter with a condition and asserts whether the condition is valid. Differently, the `assertEquals` method asserts whether two objects are equal through two mandatory parameters for the expected and actual values. According to Schmetzer's online post[2], many developers use a single assertion method to develop all the assertions they need. Commonly, developers pass conditional expressions as a parameter of the `assertTrue` method to check whether two objects are equal instead of using the `assertEquals` method, characterizing the Inappropriate Assertion (InA) test smell (KUMMER; NIERSTRASZ; LUNGU, 2015).

Figure 4.3 illustrates the detection and refactoring of the InA test smell we found in practice. *stmt* is a set of optional statements, a and b are the expected and actual values

[2] Available at: <https://exubero.com/junit/anti-patterns/>

of an object, and M is an optional message to describe the assertion. It is important to note that JUnit5 receives M as the last parameter, while previous versions of the testing framework receive M as the first parameter.

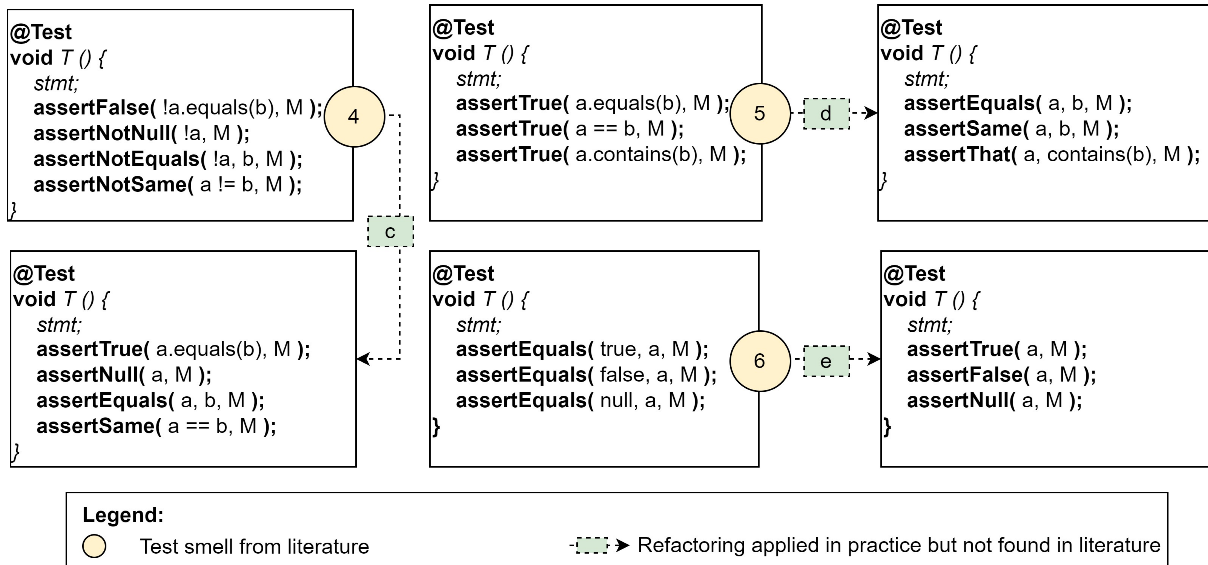


Figure 4.3: Detection and refactoring the InA test smell.

Detecting the Inappropriate Assertion test smell

(4) *Identify the `not` operator within a parameter.* For readability purposes, the `assert` methods are given in pairs to assert whether a condition is true (`assertTrue` method) or false (`assertFalse` method), two objects are equal (`assertEquals` method) or different (`assertNotEquals` method), two objects refer to the same object (`assertSame` method) or different objects (`assertNotSame` method), and an object is null (`assertNull` method) or not null (`assertNotNull` method). However, developers can be unaware of `assert` methods and create logic using the `not (!)` operator within the assertions.

(5) *Identify conditional expressions as a parameter.* The `assertTrue` and `assertFalse` methods assert whether a condition is true or false, respectively. The test smell occurs when developers force a conditional expression into the parameter of those assertions. For example, developers can create an expression with the `.equals` method and pass it as a parameter to the `assertEquals` method instead of using the `assertEquals` method to assert that two objects are equal. Similarly, developers can use the `==` operator to assert that two objects refer to the same object instead of using the `assertSame` method and other methods such as `.contains` to match values instead of using the `assertThat` method.

(6) ***Identify reserved words as parameters.*** The `assertEquals` method asserts whether two objects are equal, and the `assertNotEquals` method asserts whether they are different. The `assertSame` and `assertNotSame` methods assert whether two objects refer or do not to the same object, respectively. Such methods require two mandatory parameters (expected value and actual value). The test smell occurs when developers pass a reserved word as the expected value in those methods to verify whether the actual value equals `true`, `false`, or `null`.

Refactoring the Inappropriate Assertion test smell

(c) ***Replace the `not (!)` operator within the assertions with an appropriate assertion.*** Instead of creating a conditional logic in the assertions using the `not (!)` operator, developers should choose the corresponding pair of the specific `assert` method. The refactoring consists of the following operations:

1. Replace the `assert` method with its respective pair. More specifically:
    - the `assertTrue` method by the `assertFalse` method, or vice-versa;
    - the `assertNull` method by the `assertNotNull` method, or vice-versa;
    - the `assertEquals` method by the `assertNotEquals` method, or vice-versa;
    - the `assertSame` method by the `assertNotSame` method, or vice-versa;
    - the `assertNull` method by the `assertNotNull` method, or vice-versa.
2. Remove the `not (!)` operator of the a or b corresponding parameter in the assertion.
3. Add the M explanation message as an optional parameter.

(d) ***Split conditional expressions into two different parameters.*** The refactoring consists of splitting the conditional expression of the `assertTrue` or `assertFalse` methods into two parameters for an adequate `assert` method. The refactoring consists of the following operations:

1. Replace the `assert` method with:
    - If the assertion checks whether an a object equals a b object using the `.equals` method, replace the assertion with the `assertEquals` method;
    - If the assertion checks whether an a object refers to the same b object using the `==` operator, replace the assertion with the `assertSame` method;
    - If the assertion checks whether an a object matches the b object using the `.contains` method, replace the assertion with the `assertThat` method;

2. Split the conditional expression to pass the a and b objects as two mandatory parameters of the `assertEquals` or `assertSame` methods.
3. Add an M explanation message as an optional parameter.

(e) *Replace the reserved words with an appropriate assertion.* JUnit supports specific `assert` methods for reserved words. The refactoring consists of the following operations:

1. Replace the `assert` method with:
    - If the assertion of the b expected value equals `true`, replace the assertion with the `assertTrue` method;
    - If the assertion of the b expected value equals `false`, replace the assertion with the `assertFalse` method;
    - If the assertion of the b expected value equals `null`, replace the assertion with the `assertNull` method;
2. Pass the a actual value as a mandatory parameter of the `assertTrue`, `assertFalse`, or `assertNull` methods;
3. Add an M explanation message as an optional parameter.

It is worth noting that developers can combine those refactoring operations to address more complicated inappropriate assertions. For example, an `assertTrue` method verifies whether the a value equals true. First, the developer splits the conditional expression into two different parameters using the `assertEquals` method (5-d), then replaces the assert method with the `assertTrue` method according to its reserved word (6-e).

Samples from practice

Replacing the `assertTrue` method with the `assertFalse` method (4-c). Listing 4.5 presents the `testGenerateClientId` test method of the `Kafka` project (APACHE KAFKA, 2021b). It uses an `assertTrue` method to check whether the `ids.contains(id)` condition is false (line 293, highlighted in red). Refactoring the assertion consists of removing the `not (!)` operator of the conditional expression and replacing the `assertTrue` method with the `assertFalse` method (line 293, highlighted in green). In addition, the developer could use the `assertThat` method instead of the `.contains` method.

Replacing the `assertTrue` method with the `assertEquals` method (5-d). Listing 4.6 presents the `testPhases` test method of the `Cxf` project (APACHE CXF, 2019c). It contains an `assertTrue` method checking whether `cxfPhases` and `defaultPhases` ob-

```
@@ -290,7 +290,7 @@ public void testGenerateClientId() {
290 290     Set<String> ids = new HashSet<>();
291 291     for (int i = 0; i < 10; i++) {
292 292         String id = KafkaAdminClient.generateClientId(newConfMap(AdminClientConfig.CLIENT_ID_CONFIG, ""));
293     -       assertTrue(!ids.contains(id), "Got duplicate id " + id);
    293 +       assertFalse(ids.contains(id), "Got duplicate id " + id);
294 294         ids.add(id);
295 295     }
```

Listing 4.5: Diff between the smelly and refactored `testGenerateClientId` method of the `KafkaAdminClientTest` class of the `Kafka` project (APACHE KAFKA, 2021b).

```
@@ -168,11 +168,11 @@ public void testPhases() {
168 168     SortedSet<Phase> cxfPhases = cxfPM.getInPhases();
169 169     SortedSet<Phase> defaultPhases = defaultPM.getInPhases();
170 170     assertEquals(defaultPhases.size(), cxfPhases.size());
171     -   assertTrue(cxfPhases.equals(defaultPhases));
    171 +   assertEquals(cxfPhases, defaultPhases);
172 172     cxfPhases = cxfPM.getOutPhases();
173 173     defaultPhases = defaultPM.getOutPhases();
174 174     assertEquals(defaultPhases.size(), cxfPhases.size());
175     -   assertTrue(cxfPhases.equals(defaultPhases));
    175 +   assertEquals(cxfPhases, defaultPhases);
176 176 }
```

Listing 4.6: Diff between the smelly and refactored `testPhases` method of the `SpringBusFactoryTest` class of the `Cxf` project (APACHE CXF, 2019c).

jects are equal (lines 171 and 175, highlighted in red). Refactoring consists of passing the two objects as parameters in the `assertEquals` method (lines 171 and 175, highlighted in green).

Replacing the `assertEquals` method with the `assertTrue` method (6-e). Listing 4.7 presents the `testSoapHeader` test method of the `Cxf` project (APACHE CXF, 2019a). It uses an `assertEquals` method with a `true` value as a parameter (line 879, highlighted in red). Refactoring consists of replacing the `assertEquals` method with an `assertTrue` method and removing the unnecessary parameter for fixing the test smell (line 879, highlighted in green).

(e) ***Replacing the `assertTrue` method with the `assertNotNull` method (4-c)(5-d)(6-e)*** Listing 4.8 presents the `testValidSecurityContextToken` test method of the `Cxf` project (APACHE CXF, 2019a). It uses an `assertTrue` method to check whether

```
...    @@ -877,7 +877,7 @@ public void testSoapHeader() throws Exception {
877 877     WebParam webParamAnno = AnnotationUtil.getWebParam(method, "SOAPHeaderInfo");
878 878     assertEquals("INOUT", webParamAnno.mode().name());
879     -   assertEquals(true, webParamAnno.header());
    879 +   assertTrue(webParamAnno.header());
880 880     assertEquals("header_info", webParamAnno.partName());
...
```

Listing 4.7: Diff between the smelly and refactored`testSoapHeader` method of the `CodeGenTest` class of the `Cxf` project (APACHE CXF, 2019a).

```
...    @@ -83,8 +83,8 @@ public void testValidSecurityContextToken() throws Exception {
83    -   assertTrue(
84    -       validatorResponse.getAdditionalProperties().get(SCTValidator.SCT_VALIDATOR_SECRET) != null
   83 +   assertNotNull(
   84 +       validatorResponse.getAdditionalProperties().get(SCTValidator.SCT_VALIDATOR_SECRET)
85 85     );
...
```

Listing 4.8: Diff between the smelly and refactored `testValidSecurityContextToken` method of the `SCTValidatorTest` class of the `Cxf` project (APACHE CXF, 2019a).

an object is not null (lines 83 and 84, highlighted in red). Refactoring combines all the operations we listed above (lines 83 and 84, highlighted in green): i) replacement of the `assertTrue` method with the `assertFalse` method and removal of the `not (!)` operator (4-c), ii) split the conditional expression of the `assertFalse` method in two parameters for the `assertEquals` method (5-d), and iii) replacement of the reserved word with the `assertNull` method (6-e).

4.3.3 The Assertion Roulette test smell

According to Meszaros (2007), test methods should verify a single test condition. Conversely, Deursen *et al.* (2001) defined the Assertion Roulette (AR) test smell as a collection of assertions without an explanation message within a single test method. Undocumented assertions that can check different conditions make it hard for developers to trace which one indicates a problem.

Figure 4.4 presents the detection and refactoring of the AR test smell. In that figure, $stmt$ and $stmt'$ are a set of statements, A is a set of assertions that comprises all $assert$ methods implemented by the $Assert$ class of JUnit, and M is a set of optional messages to describe $assert$ methods.

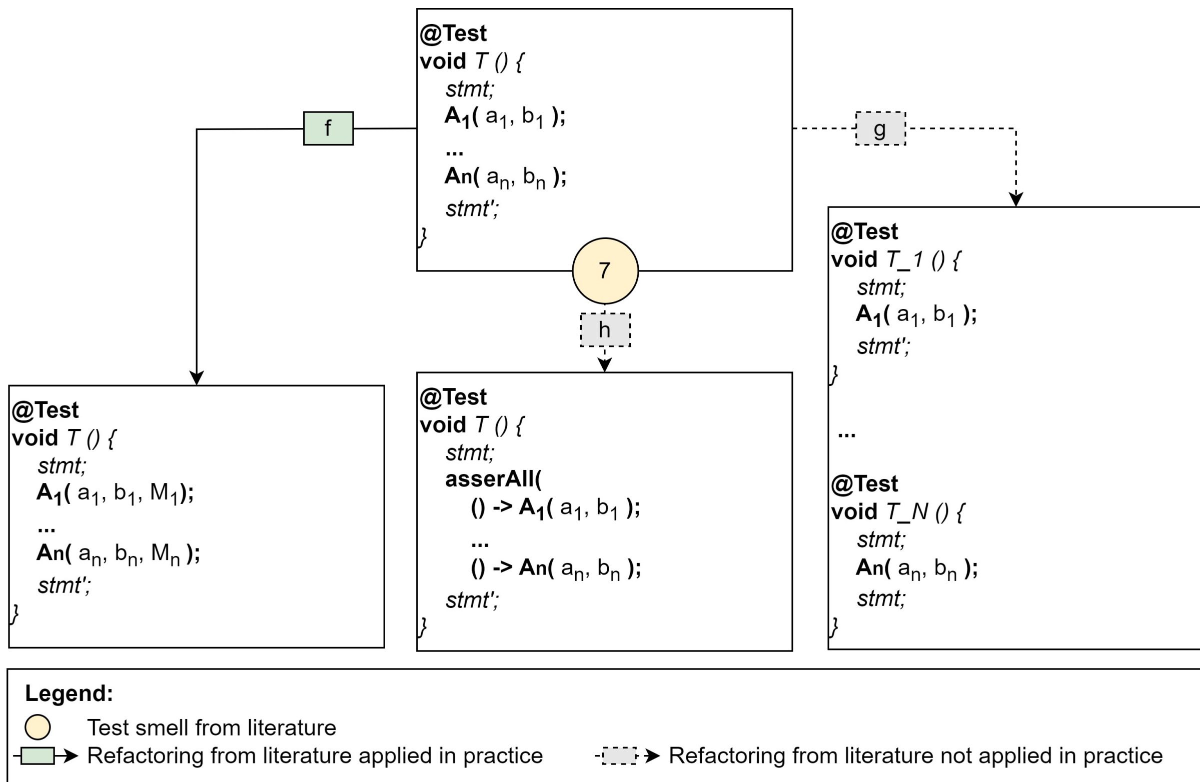


Figure 4.4: Detection and refactoring the AR test smell.

Detecting the Assertion Roulette test smell

(7) ***Identify undocumented assertions in a test method.*** According to Peruma *et al.* (2019), detecting the AR test smell consists of identifying multiple assertion statements in a test method without a descriptive message. Existing detection tools implement different rules to detect that test smell. For example, the `tsDetect` tool verifies whether a test method contains more than one assertion statement without an explanation message, and the `JNose Test` tool counts all of them as a test smell.

Refactoring the Assertion Roulette test smell

(f) ***Add an explanation message to the assertions.*** Deursen *et al.* (2001) suggested using the optional first parameter of the `assert` methods of JUnit to give an explanatory message to the user when the assertion fails. Figure 4.4 presents a T test method that contains an optional set of *stmt* statements, followed by a set of n sequential A assertions, and finalized by a set of optional $stmt'$ statements. The set of A assertions comprises all *assert* methods implemented by the *Assert* class of JUnit. Refactoring consists of adding an M message as an optional parameter in the set of A assertions. The existing

test refactoring recommendation tools have implemented that refactoring operation to fix the AR test smell, converging with the developers' practices.

(g) *Split the assertions into single test methods.* Meszaros (2007) suggested splitting the test method into single-condition tests with more than one assertion statement. Figure 4.4 presents a T test method that contains an optional set of $stmt$ statements, followed by a set of n sequential A assertions, and finalized by a set of optional $stmt'$ statements. The set of A assertions comprises all $assert$ methods implemented by the $Assert$ class of JUNIT. Refactoring consists of splitting the T test method into a set of $T_1 ... T_n$ test methods, each one comprising a single assertion.

(h) *Surround assertions with `assertAll` method.* Soares *et al.* (2022) proposed using the `assertAll` method, a feature specific to JUNIT5, to group all the assertions without explanation of a test method. Figure 4.4 presents a T test method that contains an optional set of $stmt$ statements, followed by a set of n sequential A assertions, and finalized by a set of optional $stmt'$ statements. The set of A assertions comprises all $assert$ methods implemented by the $Assert$ class of JUNIT. Refactoring consists of using lambdas expressions $() - >$ to pass the set of assertions as a parameter of the `assertAll`.

Samples from practice

Adding an explanation message as an optional parameter (7-f). Listing 4.9 presents refactoring for a group of undocumented assertions (lines 170, 173, and 176 highlighted in red) in the `MirrorSourceConnectorTest` test method from the `Kafka` project (APACHE KAFKA, 2021a). Given that the test code is in JUNIT5, refactoring consists of adding a third optional parameter with the explanation messages (lines 170, 173, and 176 highlighted in green).

```
...       @@ -169,13 +169,13 @@ public void testMirrorSourceConnectorTaskConfig() {
169  169      Map<String, String> t1 = output.get(0);
170      -    assertEquals("t0-0,t0-3,t0-6,t1-1", t1.get(TASK_TOPIC_PARTITIONS));
     170 +    assertEquals("t0-0,t0-3,t0-6,t1-1", t1.get(TASK_TOPIC_PARTITIONS), "Config for t1 is incorrect");
171  171
172  172      Map<String, String> t2 = output.get(1);
173      -    assertEquals("t0-1,t0-4,t0-7,t2-0", t2.get(TASK_TOPIC_PARTITIONS));
     173 +    assertEquals("t0-1,t0-4,t0-7,t2-0", t2.get(TASK_TOPIC_PARTITIONS), "Config for t2 is incorrect");
...
```

Listing 4.9: Diff between the smelly and refactored `testMirrorSourceConnectorTaskConfig` method of the `MirrorSourceConnectorTest` class of the `Kafka` project (APACHE KAFKA, 2021a).

4.3.4 The Bad Naming test smell

Software projects with test suites written in JUnit should follow the naming conventions as pre-pending or appending the word "Test" to the name of the production class under test in the same package hierarchy (PERUMA *et al.*, 2020a). For example, a production class in the `/src/java/example/` package is called `ExampleName.java`, so its test class should be in the `/src/test/example` package and named `ExampleNameTest.java` or `TestExampleName.java`.

Figure 4.5 presents the detection and refactoring of the Bad Naming (BaN) test smell. The test class name should be the production class name with the word "Test" in it and be in the same package hierarchy (e.g., `src/main/ProductionClassName` and `src/test/ProductionClassName`). Therefore, the refactoring operation changes the name of the test class by adding the prefix or suffix `Test`.

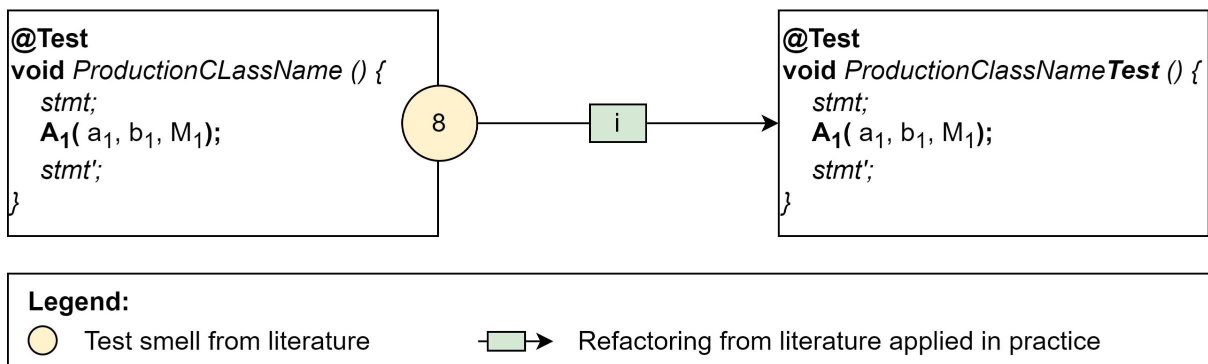


Figure 4.5: Detection and refactoring the BaN test smell.

Detecting Bad Naming test smell

(8) ***Identify test classes' names without the word "Test" in it.*** The detection consists of identifying classes without the word `Test` in their names. The test classes should be located at the `/test/` package and contain test methods with the `@Test` annotation.

Refactoring Bad Naming test smell

(i) ***Rename test classes.*** The refactoring consists of pre-pending or appending the `Test` word in the test class name.

Samples from practice

Renaming the test class (8-i). Listing 4.10 presents the `ProducerUploadFile` test class of the `Camel` project (APACHE CAMEL, 2019b). The class does not follow the naming convention of JUnit (line 35, highlighted in red). Its refactoring consists of renaming the class to `ProducerUploadFileTest` by appending the `Test` word (line 35, highlighted in green).

```
...        @@ -33,7 +33,7 @@
33    33   import org.junit.Test;
34    34
35       - public class ProducerUploadFile extends SoroushBotTestSupport {
      35 + public class ProducerUploadFileTest extends SoroushBotTestSupport {
...
```

Listing 4.10: Diff between the smelly and refactored `ProducerUploadFile` class of the `Camel` project (APACHE CAMEL, 2019b)

4.3.5 Other test smells

Our analysis found five additional test smells within our sample, although they occurred less frequently. Some test smells, such as removing a single line or method, were relatively simple to resolve. We chose non-provide detailed descriptions of those straightforward cases to avoid redundancy. However, other refactorings proved more intricate, demanding a deeper comprehension of the production code. In light of that complexity, we avoid delineating the specific strategies for those intricate refactorings, acknowledging that standardized procedures might inadequately encompass their contextual nuances.

Figure 4.6 illustrates the detection and refactoring of various test smells. In that figure, the circle symbolizes the detection of test smells, and the square represents the corresponding refactoring operations. In that visual representation, *stmt* and *stmt′* denote sets of statements, A refers to a set of assertions, and M represents a set of optional messages. The specific test smells involved are elaborated upon in Peruma et al.'s paper (PERUMA *et al.*, 2019). We explained them next.

Ignored Test (IgT). It occurs when suppressing the test methods of the running. The ignored test methods result in overhead concerning compilation time (PERUMA *et al.*, 2019).

(9) **Identifying ignored methods.** The detection consists of identifying classes or methods with the `@Disabled` annotation in JUnit5 or `@Ignored` annotation in

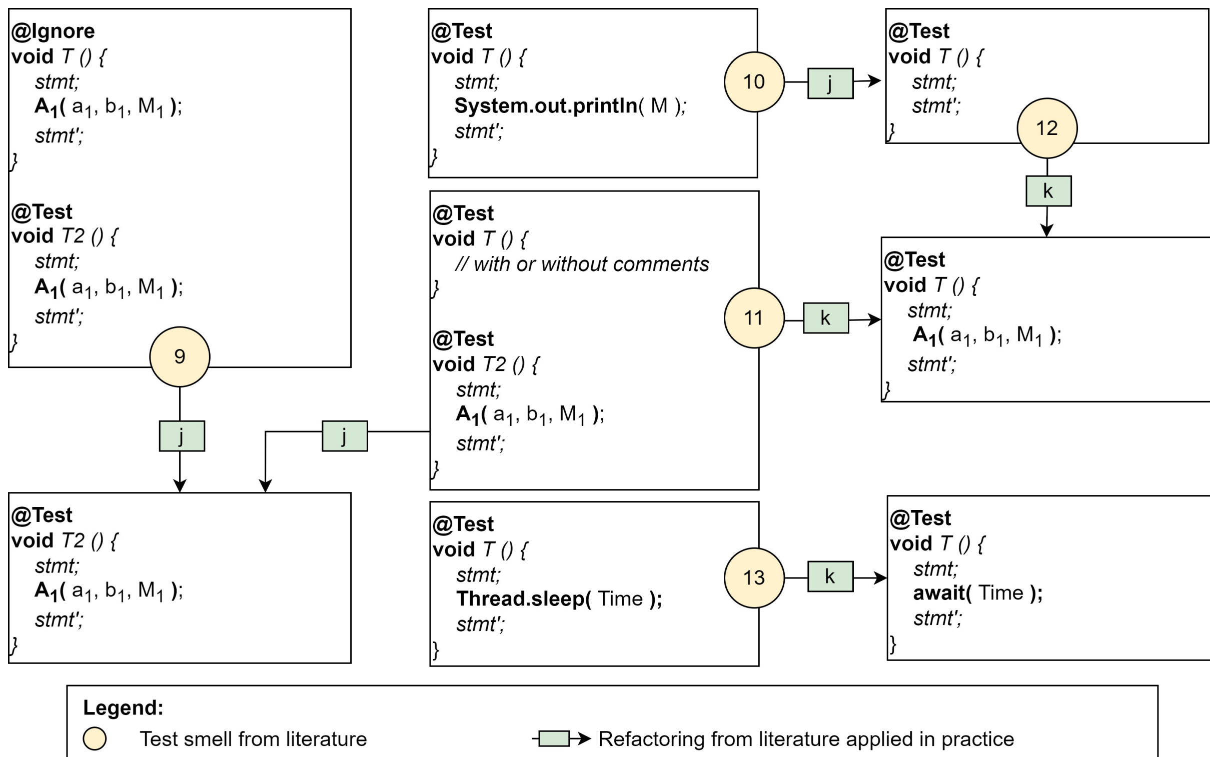


Figure 4.6: Detection and refactoring the other test smell.

previous versions of the JUnit testing framework.

(j) **Refactoring by code removal.** We found the fixing of two instances of the IgT test smell by removing the test method with the `@Disabled` or `@Ignored` annotation from the test suite.

Redundant Print (RP). It occurs when test methods use print statements. The unit tests are executed as part of an automated script, making the print statements redundant (PERUMA *et al.*, 2019).

(10) **Identifying calls for the `System.out.print` method.** The detection consists of identifying lines where developers call the `System.out.print` method of Java to print the test methods output instead of using assertions.

(j) **Refactoring by code removal.** We found the fixing of two instances of the RP test smell by removing the lines with a call to the `System.out.println` method.

Empty Test (EpT). It occurs when a test method has no executable statements. As the test method does not assert any condition, JUnit always indicates that the test

passes (PERUMA *et al.*, 2019).

(11) **Identifying methods with no executable body.** The detection consists of identifying test methods containing an empty or commented body, i.e., with no executable statements.

(j) **Refactoring by code removal.** We found the fixing of two instances of the EpT test smell by removing the test method.

(k) **Refactoring by code addition.** We found the fixing of two instances of the EpT test smell by completing the body of the test method.

Unknown Test (UT). It occurs when a test method has an executable body but no assertions. JUnit shows the test method as passing if the statements within the test method did not result in a thrown exception when executed (PERUMA *et al.*, 2019).

(12) **Identifying methods with executable body but no assertions.** The detection consists of identifying test methods that do not assert any condition while meeting the criteria: (i) contain an executable body, (ii) do not throw an exception, (iii) it is not ignored, and (iv) do not call the `System.out.print` method.

(k) **Refactoring by code addition.** the fixing of 50 instances of the UT test smell by completing the body of the test method.

Sleepy Test (ST). It occurs when a test method needs to pause its execution for a certain duration and then continue its execution. Depending on the testing environment, it can lead to unexpected results (PERUMA *et al.*, 2019).

(13) **Identifying `Thread.sleep` method.** The detection consists of identifying test methods that call the `Thread.sleep` method to force the test code to wait for some result and then proceed with its execution.

(k) **Refactoring by code addition.** We found the fixing of 12 instances of the ST test smell by writing a new logic to replace the `Thread.sleep` method. Often, the new solution involved the Java `Awaitility` library.

Samples from our dataset

Removal of the unused test method that contains the `@Ignore` annotation (9-j). Listing 4.11 presents the `InMemoryMapTest` test class of the `Camel` project (APACHE ACCUMULO, 2019). It contains the ignored method `parallelWriteSpeed` using the `@Ignore` annotation of JUnit4. As the test method was non-used, removing it from the test class (lines 516 - 554) was the solution.

```
        @@ -516,40 +516,0 @@ static long sum(long[] counts) {
516  -      // - hard to get this timing test to run well on apache build machines
517  -      @Test
518  -      @Ignore
519  -      public void parallelWriteSpeed() throws Exception {
 ...                                  ...
554  -      }
 ...
```

Listing 4.11: Diff between the smelly and refactored `InMemoryMapTest` class of the `Camel` project (APACHE ACCUMULO, 2019).

Removal of calls for the `System.out.print` method (10-j). Listing 4.12 presents `idtestMultipleMessageConsumedByClusterwithConcurrentConfiguration` method of the `PulsarConcurrentConsumerInTest` test class from the `Camel` project (APACHE CAMEL, 2020a), which has the RP test smell. The solution was removing calls for the `System.out.print` method that prints the communication status at the test method (lines 108 and 110, highlighted in red).

```
 ...          @@ -108,6 +108,4 @@ public void testMultipleMessageConsumedByClusterwithConcurrentConfiguration() th
108       -      System.out.println(NUMBER_OF_CONSUMERS + " messages sent, waiting for receival");
109   108        MockEndpoint.assertIsSatisfied(10, TimeUnit.SECONDS, to);
110       -      System.out.println("Messages received");
111   109
112   110        producer.close();
113   111    }
 ...
```

Listing 4.12: Diff between the smelly and refactored `PulsarConcurrentConsumerInTest` class of the `Camel` project (APACHE CAMEL, 2020a).

Removal of methods with no executable body (11-j). Listing 4.13 presents the EpT test smell in the `build` method of the `BigDecimalFormatFactoryTest` test class from the `Camel` project (APACHE CAMEL, 2020b). The solution was removing the entire method as it does not contain a body (lines 44 - 48, highlighted in red).

```
...     @@ -42,8 +42,2 @@ public void canBuild() throws Exception {
42  42  }
43   -
44   -    @Test
45   -    public void build() throws Exception {
46   -
47   -    }
48   -
49  43  }
```

Listing 4.13: Diff between the smelly and refactored `BigDecimalFormatFactoryTest` class of the `Camel` project (APACHE CAMEL, 2020b).

```
49  50  @Test
50   -  public void testInvalidHostConfiguration() throws Exception {
51   -      // dummy
    51 + public void testInvalidHostConfiguration() {
    52 +     ResolveEndpointFailedException cause = assertIsInstanceOf(ResolveEndpointFailedException.class,
           exception.getCause());
    53 +     assertTrue(cause.getMessage().endsWith("Unknown parameters=[{xxx=true}]"));
52  54  }
...
```

Listing 4.14: Diff between the smelly and refactored `HttpInvalidHttpClientConfigurationTest.java` class of the `Camel` project (APACHE CAMEL, 2020b).

Addition of an executable body to empty methods (11-k). Listing 4.14 presents the `testInvalidHostConfiguration` method from the `Camel` project (APACHE CAMEL, 2020b), which has the EpT test smell. It does not contain an executable body (lines 50 and 51, highlighted in red). The solution was adding a body to the test method (lines 51 - 53, highlighted in green).

Addition of assertion to test methods without assertions (12-k). Listing 4.15 presents the `testAcceptVariantString` method of the `ServerLocalTest` test class from the `Camel` project (APACHE CAMEL, 2020b). The test method has an executable body but no assertions, corresponding to UT test smell (line 66, highlighted in red). The solution was identifying which `assert` method and the production code it should verify (line 67, highlighted in green).

Replacing the test code logic to remove the `Thread.sleep` (13-k). Listing 4.16 presents the *Sleep Thread* test smell in the `testDefaultTimeoutMapPurge` method of the `DefaultTimeoutMapTest` test class from the `Camel` project (APACHE CAMEL, 2021a).

```
...    @@ -64,5 +65,3 @@ public void shouldStartComponent() {
64  65   @Test
65  66   public void testAcceptVariantString() {
66     -     sendBody(MILO_ITEM_1, new Variant(0.0));
    67 +     Assertions.assertDoesNotThrow(() -> sendBody(MILO_ITEM_1, new Variant(0.0)));
67  68   }
...
```

Listing 4.15: Diff between the smelly and refactored `ServerLocalTest` class of the `Camel` project (APACHE CAMEL, 2020b).

```
       @@ -58,13 +59,8 @@ public void testDefaultTimeoutMapPurge() throws Exception {
58  59     map.put("A", 123, 50);
59  60     assertEquals(1, map.size());
60  61
61     -   Thread.sleep(250);
62     -   if (map.size() > 0) {
63     -       LOG.warn("Waiting extra due slow CI box");
64     -       Thread.sleep(1000);
65     -   }
66     -
67     -   assertEquals(0, map.size());
    62 +   await().atMost(Duration.ofSeconds(2))
    63 +       .untilAsserted(() -> assertEquals(0, map.size()));
    64
    65     map.stop();
...
```

Listing 4.16: Diff between the smelly and refactored `DefaultTimeoutMapTest` class of the `Camel` project (APACHE CAMEL, 2021a).

The test method has a conditional statement combined with the use of the `Thread.sleep` method to slow down the test method execution (lines 61 - 67, highlighted in red). The solution was removing the entire logic, evolving the `Thread.sleep`, and adding one using the `await` method (lines 62 and 63, highlighted in green). We noticed the `await` method often replaces the `Thread.sleep` method, but we could not establish a pattern to refactor the test code logic.

4.4 A COMPARATIVE AND PRACTICAL ANALYSIS OF THE CATALOG

To answer $\mathbf{RQ}_{1.2}$, we analyzed our catalog of test refactorings compared to the ones proposed in the literature and the practices adopted on open-source projects.

4.4.1 How our catalog of test smell refactorings compares to the state-of-the-art?

This section discusses our findings in comparison with the state-of-the-art. Figure 4.7(a) shows the test smells considered in previous studies (manually or automatically refactored) compared to the ones in our catalog. Figure 4.7(b) shows the test smell refactoring strategies considered in previous studies compared to the ones in our catalog.

Test smell detection and refactoring tools.

Many tools exist in the literature for detecting and refactoring test smells. Aljedaani *et al.* (2021) conducted a systematic mapping study that reported 22 peer-reviewed tools for test smell detection, with only four of them supporting refactorings for JUnit, as follows:

- RTj is a framework that analyzes test cases developed with JUnit4 to detect and refactor test cases that always pass (MARTINEZ *et al.*, 2020);
- TestHound is a tool that detects test smells related to the fixture strategies used in test code developed with JUnit or TestNG (GREILER; DEURSEN; STOREY, 2013);
- DARTS is an IntelliJ plug-in to detect and refactor non-cohesive test methods written with JUnit (version independent) (LAMBIASE *et al.*, 2020);
- RAIDE is an Eclipse plug-in to detect and refactor duplication and lack of documentation in assert methods (SANTANA *et al.*, 2020).

However, according to Figure 4.7 (a&b), we observed that refactoring recommendation tools only support some of the test smells and the test refactorings.

Test smell refactoring catalogs.

Several catalogs of test smell refactoring strategies exist in the literature.

- Deursen *et al.* (2001) proposed the first catalog of test smells and a variant of an existing refactoring from Fowler's catalog or one specific test-refactoring;
- Peruma *et al.* (2019) proposed an extended catalog of test smells with strategies to prevent their insertion in the test code;

- Kim, Chen and Yang (2021) utilized existing catalogs to identify test refactorings applied in practice to fix test smells;
- Differently from the above catalogs, which focused on JUnit4 and earlier versions, Soares *et al.* (2022) investigated test smells and refactorings that can occur with the newly introduced features of JUnit5.

Though JUnit constructs developers use are syntactically and semantically valid, they do not guarantee the absence of test smells in test suites. Hence, testing frameworks evolve to enhance testing practices, allowing developers to create robust, maintainable, and smell-free test suites. For example, the `@Test` annotation with the `expected` parameter and the `@Rule` annotation are valid constructs when using JUnit4. However, when migrating to JUnit5, those constructs would correspond to potential test smells since JUnit5 introduced a (`assertThrows`) method for exception handling, making the previous methods less relevant. That shift illustrates how the evolution of testing frameworks can lead to certain test smells becoming obsolete or less appropriate.

Interestingly, as Figure 4.7 (a-b) shows, we observe a test smell that received little attention in the literature (the InA test smell). However, that test smell showed great importance to developers, primarily inferred by the high acceptance and appreciation of our refactoring of this particular smell, as indicated by the results of our submitted pull requests. Besides, we found three different refactoring strategies for that test smell not reported in prior research.

Test smell refactorings with varying complexity and context dependency.

Our dataset includes test smells that vary in the complexity of their refactorings, ranging from straightforward to complex, each requiring context-specific strategies. The more complex refactorings are particularly challenging due to their context-sensitive nature, encouraging future research to develop standardized procedures that effectively capture this context dependency. For example, when refactoring the ST test smell, it is important to ensure the modification considers the test logic and does not change it since the `Thread.sleep` method can force the test method execution to slow down to wait for or simulate an event.

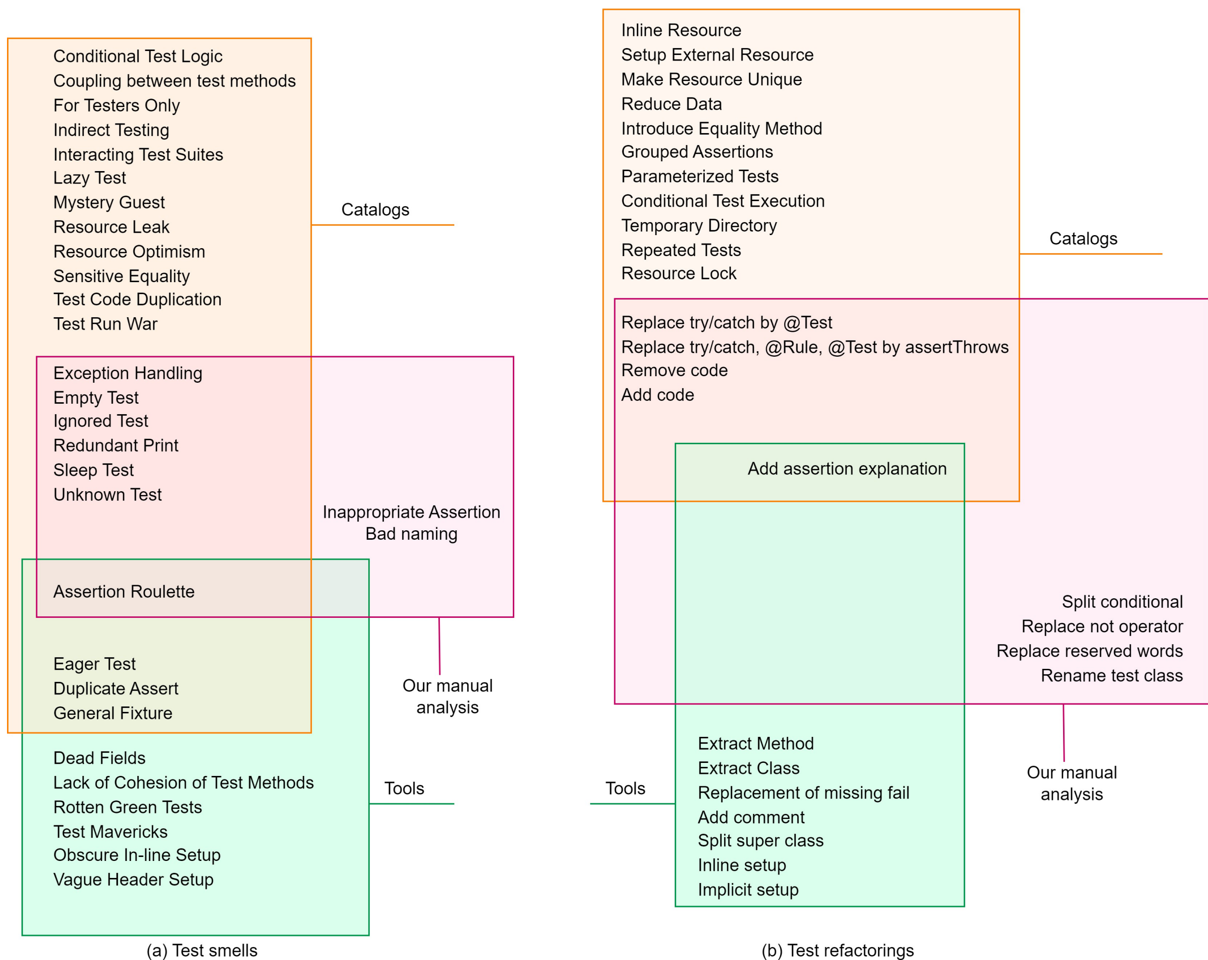


Figure 4.7: Comparison between our findings and the state-of-the-art regarding test smells and test refactorings.

4.4.2 How our catalog of test smell refactorings is acceptable in practice?

The main goal of $RQ_{1.2}$ is to explore the refactorings that developers implement in practice to gain a deeper understanding of the common challenges they face and the methods they currently employ to refine test code. To enrich our insights beyond our automated and manual empirical analyses, we sought feedback from software developers who have contributed to the projects included in our dataset to allow gathering their perspectives regarding several aspects: their awareness of test smells within their projects, their acceptance of the test smell refactorings we identified, and their views on the reasons behind the existence of such test smells. Additionally, we sought their opinions on how useful

the refactorings are in their software development and testing process.

We adopted a proactive and collaborative approach further to understand test smells and their refactorings in practice to achieve the above objectives. In particular, we submitted pull requests to the 13 GitHub projects included in our dataset, proposing refactorings based on our empirical findings. We randomly selected samples of different test smell types that still exist in those projects and applied refactorings manually on their remote repositories on GitHub. That hands-on approach allowed us to assess the acceptability and practical implications of our identified refactorings and gather feedback on those refactorings by directly interacting with the software development community.

Submitting pull requests was only the first step in our engagement, as we provided developers context about our research and requested their opinion about the test smells and refactorings under study. Post-submission, we actively followed up with the developers, engaging in discussions about the changes we proposed. Developers often provided justifications not only for why certain test smells existed but also for why certain refactorings were not worth the change, which offered historical or functional insights that might take time to be apparent from data analysis. Moreover, those discussions often led to further improvements and refinements to the changes beyond our initial suggestions, which refactored more instances of certain test smells in some projects. Through that iterative and interactive process, we validated our research and contributed directly to enhancing test code quality in active projects.

We submitted 13 pull requests, one for each project we studied. We addressed one to two test smells in each pull request, in which we refactored multiple instances of each test smell. More specifically, we refactored (i) 95 instances related to the Inappropriate Assertion test smell in nine projects, (ii) 24 instances related to the ECT test smell in six projects, and (iii) 24 instances related to the AR test smell in 3 projects. Overall, 10 of the 13 pull requests (77%) have successfully been merged by developers with positive responses, with one pull request already being approved and receiving a response, but we still need to merge them. Two pull requests are still open but we received responses from developers.

We performed a qualitative analysis of the comments and categorized them into themes to better understand the sentiments of developers and their reasons for accepting or rejecting the refactorings. Our analysis of the feedback provided valuable insights into the perceptions and challenges of test smells in open-source projects. It can inform future research and tool development efforts to improve test quality. We summarize our analysis

in the following.

- **Acknowledgment of improvement (*5 comments*):** Developers' feedback suggests a general appreciation for the efforts to improve test quality. We captured that sentiment when a developer stated: "*Thanks for your contribution cleaning up our tests. We love those!*";

- **Request for rationale (*2 comments*):** While the intent behind the pull requests was clear to some, others sought further clarity. One developer remarked: "*I was curious the rationale for why `assertThrows` should replace the annotation. After clarifying what test smells are... This was a nice thing to know!*"

- **Historical context (*2 comments*):** A developer offered a historical codebase context: "*Actually, in this case, you should probably say that they became inappropriate. When this test infrastructure was built, assertNotEquals did not even exist.*" That comment highlights how certain test smells might have originated due to the limitations or norms of the time, suggesting that developers may sometimes be behind recent advancements in testing frameworks;

- **Developer Preferences (*3 comments*):** Several developers emphasized the value of explicit failure messages. One such comment pointed out: "*The use of assertNotEquals instead of assertTrue seems fine. Those specific assertion method changes are enough to add a more detailed message from JUnit on failure.*" That feedback highlights the need for tooling that refactors code and enhances its clarity and debuggability;

- **Prior Awareness (*1 comment*):** A developer from one project mentioned: "*For the problem you noticed, the community has started the shift from JUnit4 to JUnit5+AssertJ a while ago... However, due to the huge size of the codebase, the shift may take some extra months to finish.*" That comment indicates an awareness and proactive approach to modernizing testing practices;

- **More important test smells than others (*2 comments*):** The feedback offered potential focus areas for researchers and tool developers. As one developer suggested: "*If anyone were to go near old tests, I'd target things I really don't like - assertTrue/assertFalse without error messages or detail why the test failed.*"

4.4.3 Implications for software engineering research and practice

We identified several implications for software engineering research and practice.

Awareness of test smells. Our data collection indicated developers had refactored many instances of test smells without explicitly highlighting such particular refactorings in their changes (i.e., commit messages). In addition, we could prove this by our submitted pull requests, in which we targeted projects from our dataset but observed that developers appreciated our refactorings and did not have or had little idea about certain test smells and their impact. Therefore, it is crucial to increase developers' awareness of test smells, their impact on test quality, and how to refactor them properly. Those test smells can propagate without such awareness, making future refactorings harder and potentially introducing fragility into the test suite. Raising awareness can be done through workshops, documentation, and peer reviews. If developers recognize test smells like the AR or InA, they could more proactively address them during regular development.

Extended tool support for test smell detection and refactorings. Automated tools can assist developers in identifying and refactoring test smells. Our research identified new test smells (e.g., the InA test smell) and test smells straightforward yet context-specific (e.g., the ST test smell), where existing tools cannot detect and/or refactor them. Therefore, it is important to extend existing test smell detection and refactoring tools to detect and refactor test smells, provide adequate context guidance on why a certain practice is considered a test smell, and offer suggestions for refactorings. For example, suppose a tool detects one ECT test smell. In that case, it should point it to the developers and suggest ways to refactor and ensure that such refactoring is consistent and does not change the testing logic. That context ensures developers rectify the issue and understand the reasoning behind the refactoring, reinforcing best practices.

Historical analysis of test smells. While much research exists on test smells and their refactorings, little attention has been paid to the evolution of test smells across software projects. Our results showed that test smells could emerge due to a possible upgrade to the testing framework (e.g., from JUnit4 to JUnit5), such as the ECT test smell. Understanding the origin and evolution of test smells in a codebase is invaluable. Therefore, future work should conduct extensive research to analyze the history of test code, which can provide insights into why certain smells emerge and how testing and refactoring practices evolved. Moreover, observing the refactoring trends of test smells can provide insights into developers' and researchers' testing best practices.

4.5 THREATS TO VALIDITY

This section discusses the validity threats to our results.

External Validity. We studied 13 open-source JAVA projects, with test code written with JUNIT testing framework, selected by Kim, Chen and Yang (2021). Although the projects are large in scale and cover various domains, other projects might exhibit different test smells and refactoring operations. Therefore, the results can be non-generalized to other contexts and programming languages.

Construct Validity. We selected a stratified sampling of modified test classes that follow the JUNIT naming convention. Therefore, we could have missed some test classes we did not retrieve by the regular expression used to find the word "Test" in the name of test classes. Additionally, we selected the samples from the projects over three years. Since test classes are commonplace to receive new tests, we assume that most of the changes in the test classes would not be test refactorings. Therefore, we filtered the commits with the potential to exhibit test refactorings by analyzing the commits' messages. As a result, 41.7% of the test files have test refactoring operations that fix test smells. We could have used other approaches to filter test classes with a non-negligible probability of being refactorings. For example, we could consider test classes of no-modified production classes. Finally, we based our catalog of test smells and refactorings on a manual analysis of commits performed by two coders. It could lead to inconsistency or mislabeling of the changes in the test code. To address this issue, we conducted collaborative discussions to resolve any disagreement by using standardized categories from the literature as a reference.

Internal Validity. A potential threat in this study is the manual analysis of the test code to label the test refactorings that fixed test smells. Two authors independently inspected the test refactorings of 50 test files to calculate the kappa statistic and minimize the bias, and one author inspected the remaining files. Still, we could have mislabeled some test refactorings, leading to an incorrect interpretation of the results.

Conclusion Validity. During our data collection process, we used the commit messages to distinguish the test code changes meant to refactor test smells from other evolutionary changes. The data collection process could have biased the conclusions for two main reasons. First, developers rarely document refactoring activities explicitly (WEISSGERBER; DIEHL, 2006; WEISSGERBER; BIEGEL; DIEHL, 2007), but when they do, they can misuse the term "refactoring" to indicate normal code modifications

(DI *et al.*, 2018). Second, some evolutionary changes overlap the changes for fixing test smells, distinct from pure test code refactoring. As such, our conclusions can be due to evolutionary changes other than refactoring.

4.6 CHAPTER SUMMARY

This chapter presents our empirical study to catalog test refactorings used to fix test smells. Our analysis revealed 156 test files containing 611 pairs of smelly and refactored test codes. In particular, three test-specific refactorings are associated with upgrading the JUnit version, targeting two test smells (ECT and AR). Another eight test-specific refactorings are version-agnostic concerning JUnit, targeting seven test smells (InA, AR, BaN, IgT, EpT, UT, and ST). Additionally, our findings show that developers have a common interest in specific assert methods of the testing frameworks. For example, developers can force conditional expressions or redundant parameters within assert methods, leading to the InA test smell, which remains unexplored in the literature. To enrich our insights beyond our empirical analyses, we sought feedback from software developers regarding test smells and refactorings.

In summary, this chapter provides the following contributions:

- We presented an accurate and manually validated dataset (MARTINS *et al.*, 2023b) of test-specific refactorings based on 375 test files in 13 projects, showing how developers refactor test code in practice;
- We presented a catalog, publicly available (MARTINS *et al.*, 2023c), listing the test smells observed in practice and their corresponding test-specific refactorings derived from both the literature and practice;
- We submitted 13 pull requests, utilizing our catalog, for refactoring 143 test smells in the test suites of Apache projects, reaching a 77% acceptance rate.

Chapter 5

HOW TEST REFACTORINGS AFFECT TEST CODE QUALITY

Over the last decades, researchers have been proposing automated refactoring recommenders (BAVOTA *et al.*, 2014) and investigated how refactoring relates to code quality (BOIS; DEMEYER; VERELST, 2004; CHáVEZ *et al.*, 2017; SOBRINHO; LUCIA; MAIA, 2018; AZEEM *et al.*, 2019). In particular, they identified both benefits and drawbacks of its application (DALLAL, 2015; BAQAIS; ALSHAYEB, 2020; LACERDA *et al.*, 2020), finding that, while refactoring is theoretically associated with modifications that do not affect the external behavior of source code, it can induce defects (BAVOTA *et al.*, 2012; FERREIRA *et al.*, 2018; PENTA; BAVOTA; ZAMPETTI, 2020), vulnerabilities (IANNONE *et al.*, 2023), or even code smells (TUFANO *et al.*, 2017b). Those drawbacks are mainly due to refactoring activities performed manually without the support of automated tools and interleaved with other code changes (MURPHY-HILL; BLACK, 2007). This chapter is motivated by those previous studies. On the one hand, most of them focused on production code refactoring. Therefore, we argue there is a lack of investigation into *how refactoring is applied to test code.* On the other hand, we do not know if similar effects observed in previous work can arise with test refactoring, i.e., it can have some impact on both test quality, for instance, in cases where refactoring actions target the logic of a test case. Hence, we point out a *there is limited knowledge of the effects of refactoring* on both test quality.

An improved understanding of test refactoring would have several potential benefits for research and practice. In the first place, test cases represent a crucial asset for software

dependability: developer's productivity is partly dependent on the quality of test cases (MICCO, 2017), as they help practitioners decide on whether to merge pull requests or deploy the system (GRANO *et al.*, 2020). As such, analyzing how refactoring affects test cases can significantly impact practice. Secondly, researchers have shown test code design is approached substantially differently than traditional development (MESZAROS, 2007).

For those reasons, new refactoring practices were proposed to deal with quality concerns (DEURSEN *et al.*, 2001; MESZAROS; SMITH; ANDREA, 2003). While those refactoring practices were the target of some previous investigations, researchers limited their focus to how refactoring can influence test smells, i.e., symptoms of poor test code quality (SOARES *et al.*, 2020; PERUMA *et al.*, 2020b; SOARES *et al.*, 2022). Hence, it does not comprehensively analyze the *nature* and *effects* of test refactoring. More specifically, we highlight a lack of knowledge on (1) whether developers apply test refactoring operations on test classes affected by quality concerns and (2) what the effect of refactoring is on both test case qualities.

This chapter addresses this knowledge gap by carrying out an *exploratory empirical study*. We first collected test refactoring data from the change history of open-source JAVA projects from GITHUB. Next, we combined them with data from automated instruments able to profile test code from the perspective of quality metrics and test smells. Afterward, we applied statistical analyses to address two main research goals targeting (1) whether test classes with a low level of quality, in terms of test smells and code metrics, are associated with more test refactoring, and (2) what extent the removal of test smells improve the test code quality.

The remainder of this chapter is structured as follows. Section 5.1 presents the goal and research questions explored. Section 5.2 presents the experimental design to conduct the study. Section 5.3 presents the results. Section 5.4 shows the discussion of those results. Section 5.5 lists the threats to validity.

5.1 RESEARCH QUESTIONS AND OBJECTIVES

The *goal* of the empirical study is to analyze the test refactoring operations performed by developers over the history of software projects, with the *purpose* of understanding (1) whether low-quality test classes, in terms of structural metrics and test smells, provide indications on which test classes are more likely of being refactored, and (2) as a consequence, to what extent test refactoring operations are effective in improving quality

of test classes. In other terms, we are first interested in assessing the **quantity** of test refactoring operations performed on classes exhibiting test code quality issues and, in the second place, the **quality** of the test refactoring operations applied in terms of improvements provided to test code quality. The *perspective* is researchers and practitioners interested in understanding the relationship and effects of test refactoring operations on the quality of test classes.

More specifically, our empirical investigation will first aim at addressing the following research questions (**RQ**s):

> **$\mathbf{RQ}_{2.1}$.** *To what extent are test refactoring operations performed on test classes having a low level of quality, as indicated by quality metrics and test smell detectors?*

Through $\mathbf{RQ}_{2.1}$, we aim to investigate whether the low-quality test classes are associated with more test refactoring operations. It might help us to understand whether the characteristics of the test suites trigger more refactoring operations, possibly informing researchers on (1) the factors associated with test refactoring and (2) the design of novel or improved instruments to better support developers in their activities. For instance, when discovering refactoring is not frequent on smelly test classes, it is necessary to conduct further research on the motivations leading developers to refactor test code and know as to design test smell detectors to ease the application of refactoring operations.

Upon completing that investigation, we elaborate on the impact of test refactoring, addressing the following RQ:

> **$\mathbf{RQ}_{2.2}$.** *What is the effect of test refactoring on test code quality, as indicated by quality metrics and test smell detectors?*

Through $\mathbf{RQ}_{2.2}$, we aim to extend the current knowledge on the impact of test refactoring, assessing whether the test code quality changes or remains the same after applying test refactoring operations. It is worth mentioning that addressing that research question would be important, independent of the results from $\mathbf{RQ}_{2.1}$. Indeed, regardless of the amount of refactoring operations performed on low-quality and smelly test classes, it would still be possible that the specific refactoring actions targeting those classes have any impact.

To make our argument more practical, consider the case of *Extract Method* refactoring, whose suboptimal implementation can potentially affect test code effectiveness. Given a verbose test method with several steps and assertions, refactoring enables the extraction of multiple more cohesive and focused test methods on the verification of spe-

cific conditions of production methods. However, if developers inappropriately perform such an extraction, it could potentially change the test logic and harm test effectiveness. For instance, consider the T test, which verifies the $B1$ and $B2$ branches of the M production method. In this case, an Extract Method operation is supposed to split the T test so that the $T1$ and $T2$ resulting tests target the $B1$ and $B2$ branches individually. However, whether a logical relation exists between the $B1$ and $B2$ branches, the $T2$ test will still need to pass through the $T1$ test to ensure the meeting of the logical relation: suboptimal refactoring can overlook that requirement, possibly not embedding in the $T2$ test the statements required to reach the $B1$ branch.

As such, $\mathbf{RQ}_{2.2}$ provides an orthogonal view of the matter. Also, in this case, the outcome of our investigation can have implications for research and practice. First, our findings can help researchers measure the actual and practical impact of test refactoring, driving considerations on how future research efforts should be prioritized, e.g., by favoring more research on impactful refactoring operations. Second, our results can increase the practitioner's awareness of test refactoring, possibly increasing its application in practice.

5.2 EXPERIMENTAL DESIGN

This section reports the research method we apply to address our **RQs**. Figure 5.1 depicts the steps to execute our study.

5.2.1 Context of the study

The *context* of our investigation is composed of (i) software systems, (ii) metrics collected from software systems, i.e., variables, and (iii) empirical study variables, i.e., the independent and dependent variables we statistically analyzed.

Software Systems. We considered three main criteria to select suitable software systems. First, we selected open-source projects, as we need access to change history information. Second, we relied on popular, large, and real-world projects with enough releases to collect data for statistical analysis. Third, we standardized the building process to ease dependency management and streamline build configurations across all projects. As such, we used SEART tool[1] to select open-source and non-fork projects from GitHub

[1] <https://seart-ghs.si.usi.ch/>

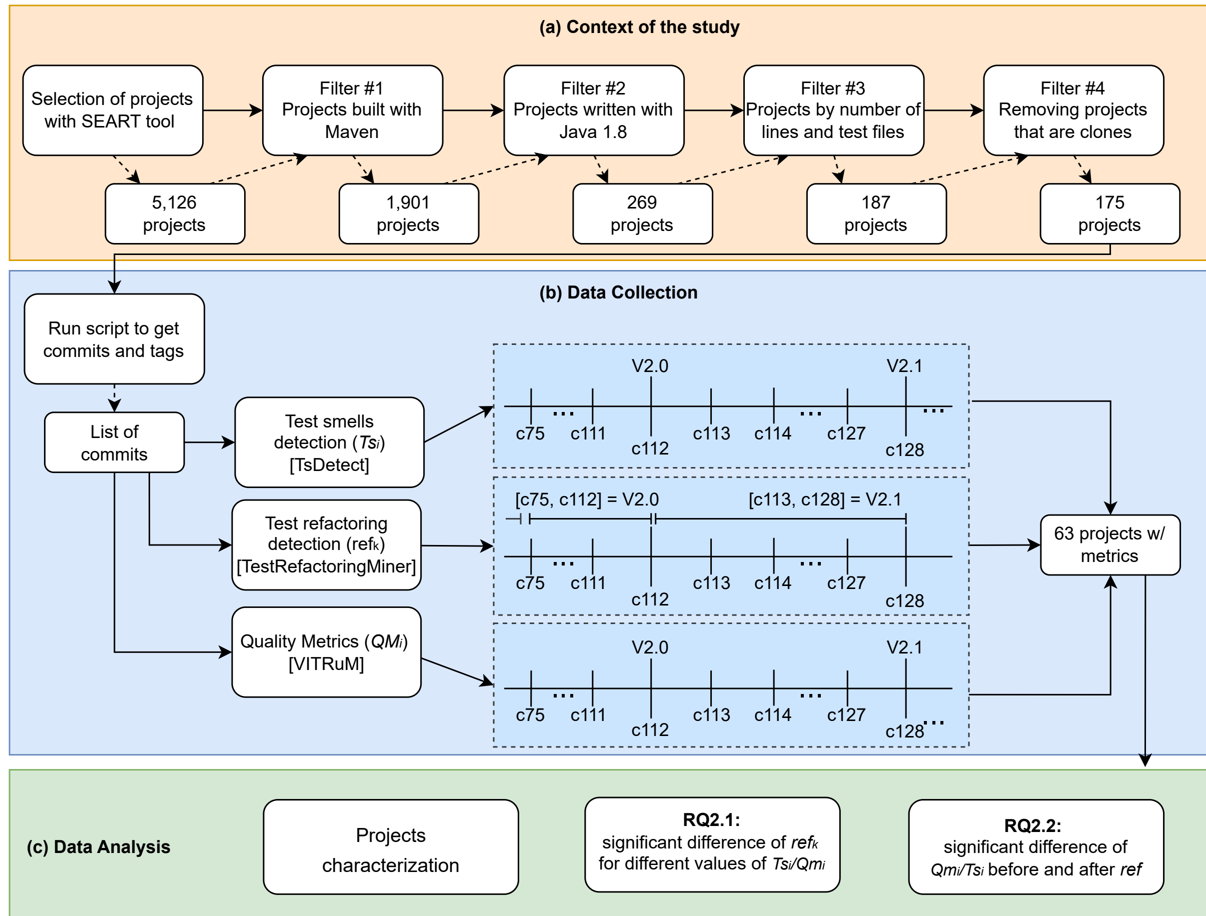


Figure 5.1: Overview of the experimental design.

with at least 100 stars, ten major releases, 1,000 lines of code, and 10 test classes. We sought JAVA projects to compile them with Maven and JAVA 8—JAVA 8 is the most popular JAVA version currently used[2]. As a result, we selected 175 projects.

Variables. We first collected test refactoring data from the change history of open-source JAVA projects from GitHub. Next, we combined them with data from automated instruments able to profile test code from the perspective of quality metrics and information on test smells.

Test Smells. In particular, we considered six test smells: 1) Assertion Roulette (AR), 2) Duplicate Assert (DA), 3) Exception Handling (ECT), 4) Eager Test (ET), 5) General Fixture (GF), and 6) Lazy Test (LT).

Test code quality. Features in that category derive from the test code attributes. In

[2] <https://www.jetbrains.com/lp/devecosystem-2021/java/>

particular, we selected metrics related to test code size, complexity, and coupling: 1) Lines of Code (LOC), 2) Number of Methods (NOM), 3) Weighted Method per Class (WMC), 4) Response for a Class (RFC), and 5) Assertion Density (AD).

Refactorings. We considered test-specific refactorings applied to fix test smells and the following refactorings from Fowlers' catalog (FOWLER, 1999): 1) Add assert explanation, 2) Extract Class, 3) Extract Method, 4) Inline Method, 5) Rename Class, 6) Replace conditional by Parameterized Test, 7) Replace NOT operator, 8) Replace Reserved Words, 9) Replace Test annotation w/ assertThrows, 10) Replace Rule annotation w/ assertThrows, 11) Replace try/catch w/ assertThrows, 12) Split Conditional Statement in Assertions, and 13) Split method.

Empirical Study Variables. In the context of $\mathbf{RQ}_{2.1}$, we are interested in assessing whether refactoring operations are more likely to be observed on test classes exhibiting test code quality concerns. As such, we defined the following empirical study variables:

Independent Variables. Those variables are factors related to the application of test refactoring, namely (i) test code quality metrics and (ii) the presence of test smells (of different types). We calculated the metrics across releases of different software systems and statistically analyzed them, as described later in this section. Multiple considerations drove the selection of those independent variables. First, we considered test code quality metrics and test smells, targets of previous research in the field (CATOLINO *et al.*, 2019; PECORELLI; PALOMBA; LUCIA, 2021), and impact test code in different manners (SPADINI *et al.*, 2018; KIM; CHEN; YANG, 2021);

Dependent Variables. Those variables are refactoring operations (of different types) observed across releases of different software systems. To select them suitably, we investigated the literature to elicit test refactoring operations previously associated with our independent variables.

When it turns to $RQ_{2.2}$, we assessed the impact of test refactoring on the test code quality. As such, we swapped independent and dependent variables: indeed, in this case, we observed how refactoring impacts test code properties rather than the opposite:

Independent Variables. Those variables are different types of refactoring operations computed across the releases of software systems considered;

Dependent Variables. Those variables are test smells and test code metrics described and computed across releases of software systems.

5.2.2 Data Collection

We used different automated tools available in the literature to extract data on quality metrics, test smells, and refactoring operations. Then, we merged the data to compose our dataset.

Collecting test code quality. To collect test code quality metrics, we executed VITRuM to calculate five static metrics from the test code (PECORELLI *et al.*, 2020b).

Collecting test smells. Among the test smell detection tools available for Java code (ALJEDAANI *et al.*, 2021), we used the tsDetect tool(PERUMA *et al.*, 2020a), the most accurate tool, with a precision score ranging from 85% to 100% and a recall score ranging from 90% to 100%. That tool performs a test code static analysis through an AST (Abstract Syntax Tree) to apply detection rules of test smells in the test files. A test file in the JUnit testing framework should follow the naming conventions of either pre-pending or appending the `"Test"` word to the name of the production class under test and at the same package hierarchy (PERUMA *et al.*, 2020a). With the detection rules, the tool can detect (i) the presence or absence of a test smell in a test class or (ii) the number of instances per test smell in a test class. In addition, it receives a configuration of the severity thresholds for each test smell (SPADINI *et al.*, 2020).

Collecting refactoring data. To detect test refactoring operations, we used the TestRefactoringMiner tool (MARTINS *et al.*, 2023a). We integrated the test detection mechanisms proposed by RefactoringMiner team in late 2021[3] and built TestRefactoringMiner on top of their refactoring mining tool. RefactoringMiner implements rules to detect 1999 refactorings from Fowler's catalog and has the highest precision (99.8%) and recall (97.6%) scores among the currently available refactoring mining tools (TSANTALIS; KETKAR; DIG, 2020). In more detail, the TestRefactoringMiner tool analyzes the added, deleted, and changed files between two project versions to detect specific test refactorings, reaching 100% precision and 92.5% recall scores. The tool operationalizes detecting all refactoring operations considered in the

[3] Available at: https://github.com/tsantalis/RefactoringMiner/pull/225

study. It is worth noting that the test refactorings consider various aspects of the test code, such as integrating new technologies like JUnit5 or improving the organization of test classes.

Data integration. Although some tools allow a finer granularity during the code analysis, they can all report the results at the class level. Therefore, we established traceability links between the test classes reported by the tsDetect, VITRuM, and TestRefactoringMiner tools. Finally, we integrated their outcome into a unique data source to be further analyzed statistically. In Figure 5.1, after executing the tools in the *Data Collection* step, the dataset remained with 63 projects because not all projects have refactorings in the test code (i.e., no intersection between the tools' outputs).

5.2.3 Data Analysis

We first formulated the working hypotheses we statistically assessed. As for $\mathbf{RQ}_{2.1}$, given a Qmi quality metric Qm_i in {LOC, NOM, WMC, RFC, AD} and a ref_k refactoring in the set of refactoring operations considered in the study, our null hypothesis is the following:

$\mathbf{Hn1}_{Qm_i - ref_k}$. *No significant difference* exists in the amount of refk performed on test classes with different Qm_i value.

As in $\mathbf{RQ}_{2.1}$, we evaluated the relation between test refactoring and test smells. Given a Ts_i test smell in the set of test smells and a ref_krefactoring in the set of refactoring operations, both considered in the study, we defined a second null hypothesis:

$\mathbf{Hn2}_{Ts_i - ref_k}$. *No significant difference* exists in the amount of ref_k performed on test classes affected and not affected by Ts_i.

As for $\mathbf{RQ}_{2.2}$, given a Qm_i quality metric, a Ts_i test smell, and a ref_k refactoring, the null hypotheses are:

$\mathbf{Hn3}_{Qm_i - ref_k}$. *No significant difference* exists in Qm_i before and after the application of ref_k.

$\mathbf{Hn4}_{Ts_i - ref_k}$. *No significant difference* exists in the number of Ts_i instances before and after the application of ref_k.

The statistical rejection of one of the null hypotheses leads us to accept the corresponding alternative hypothesis, namely:

An1$_{Qm_i-ref_k}$**.** The amount of ref_k operations on test classes having different values of Qm_i is *statistically different*.

An2$_{Ts_i-ref_k}$**.** The amount of ref_k on test classes affected and not affected by Ts_i is *statistically different*.

An3$_{Qm_i-ref_k}$**.** The Qm_i before and after the application of ref_k is *statistically different*.

An4$_{Ts_i-ref_k}$**.** The number of Ts_i instances before and after the application of ref_k is *statistically different*.

We build statistical models to verify the working hypotheses, accepting or rejecting the hypotheses.

Statistical modeling for RQ$_{2.1}$. We devised a *Logistic Regression Model* for each refactoring operation considered in the study. Such a model belongs to the class of Generalized Linear Models (GLM) (NELDER; WEDDERBURN, 1972) and relates a (dichotomous) dependent variable with either continuous or discrete independent variables. In our case, whether or not a particular type of refactoring is performed is the dependent variable, and the quality metrics are the independent variables. Before building the statistical model, we assessed the presence of multi-collinearity (O'BRIEN, 2007), which arises when two or more independent variables are highly correlated and one can predict the other. We used the `vif` (Variance Inflation Factors) function and discarded highly correlated variables with a threshold value equal to 5 (O'BRIEN, 2007).

For each statistical model, we (i) assessed whether each independent variable significantly correlates with the dependent variable (using a significance level of $\alpha = 5\%$, and (ii) quantified this correlation using the Odds Ratio (OR) (BLAND; ALTMAN, 2000), a strength measure of the association between an independent variable and a dependent variable. Higher OR values for an independent variable indicate a higher probability of explaining the dependent variable, i.e., a higher likelihood the independent variable has triggered a refactoring operation. Nonetheless, the interpretation of OR values changes depending on the different measurement scales of the independent variables, i.e., the ratio for the test code quality metrics and definite for the test smells. As for the metrics, the OR for an independent variable indicates the increment of chances for a test class to be

subject to refactoring due to a one-unit increase of the independent variable. As for test smells, the OR indicates how likely a smelly test class is involved in refactoring operations concerning a non-affected class.

The statistical significance of the correlation between independent and dependent variables allows us to accept or reject $\mathbf{Hn1}_{Qm_i-ref_k}$ and $\mathbf{Hn2}_{Ts_i-ref_k}$, while OR values will measure the strengths of the correlations.

Statistical modeling for $\mathbf{RQ}_{2.2}$. To statistically assess the impact of test refactoring on test code quality metrics and smells, we first collected all the test classes subject to the refactoring type ref_k in a generic release R_i. Afterward, we computed on the release R_i the value of test code quality metrics and test smells. Similarly, we computed on the release R_{i-1} the value of test code quality metrics and test smells. We produced two distributions: the first represents the number of test smells in R_{i-1}, i.e., before the application of ref_k; the second represents the number of test smells in R_i, i.e., after the application of ref_k. On this basis, we employed the non-parametric Wilcoxon Rank Sum Test (MCKNIGHT; NAJAB, 2010) (with α-value = 0.05), through which we accepted or rejected the null hypotheses $\mathbf{Hn3}_{Qm_i-ref_k}$ and $\mathbf{Hn4}_{Ts_i-ref_k}$.

In addition, we computed the difference between the distributions of quality metrics and test smells computed on the release R_i, and the value of the metrics and test smells computed on the release R_{i-1}. Then, we classified whether the difference between the distributions ($R_i - R_{i-1}$) was positive, negative, or neutral, indicating whether there was an improvement in the test code quality.

5.3 RESULTS

Figure 5.2 depicts the boxplots of the distributions of metrics and test smells for the sets of refactored and non-refactored tests in our dataset. It is worth mentioning all the 12,578 test classes in our dataset have refactorings from Fowler's catalog. However, we considered the test-specific refactorings with the potential to contribute to removing test smells. Therefore, we have 1,023 refactored test classes (8.1%) and 11,555 non-refactored test classes (91.8%) from that set of test-specific refactorings.

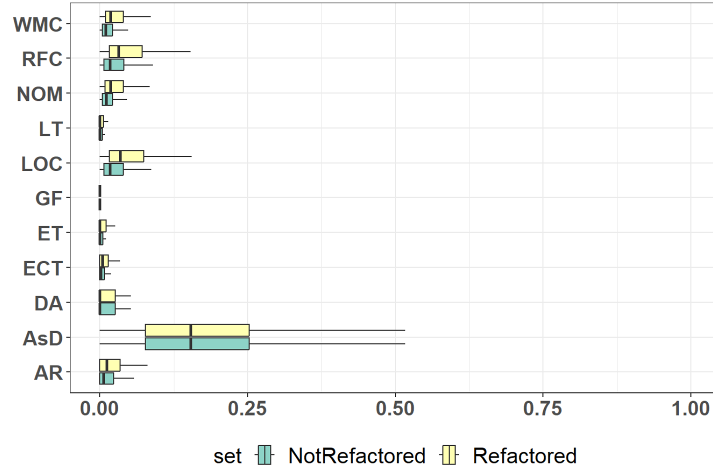


Figure 5.2: Boxplots for the distributions of metrics and test smells in the dataset.

Table 5.1: Results for the statistical model considering the quality metrics (Qm_i).

Variable	Estimate	SE	Pr(>|z|)	OR	Variable	Estimate	SE	Pr(>|z|)	OR
Add Assert Explanation					**Replace NOT operator**				
Intercept	-5.884	0.223	**2e-16**	**0.001**	Intercept	-6.519	0.305	**2e-16**	**4e-04**
NOM	2.129	2.149	0.322	8.405	WMC	-7.659	6.185	0.215	0.000
RFC	-0.112	1.865	0.952	0.894	LOC	6.345	2.416	**0.008**	**5e+01**
AsD	2.695	0.706	**0.000**	**14.799**	AsD	2.803	0.923	**0.002**	**1e+02**
Extract Class					**Replace conditional by ParameterizedTest**				
Intercept	-4.159	0.134	**2e-16**	**0.016**	Intercept	-6.322	0.364	**2e-16**	**1e-03**
WMC	0.476	1.883	0.800	1.610	LOC	7.960	4.389	0.069	2e+03
RFC	0.507	1.357	0.709	1.660	WMC	-20.544	14.810	0.165	1e-09
AsD	-1.604	0.653	**0.014**	**0.201**	AsD	1.130	1.250	0.366	3.09
Extract Method					**Replace Rule with assertThrows**				
Intercept	-2.912	0.066	**2e-16**	**0.016**	Intercept	-6.167	0.319	**2e-16**	**3e-04**
NOM	1.102	0.719	0.126	1.610	NOM	2.834	4.147	0.494	**7e+02**
RFC	2.725	0.464	**0.000**	1.660	RFC	-2.155	3.929	0.583	6e-04
AsD	-0.703	0.300	**0.019**	**0.201**	AsD	0.609	1.245	0.625	1e+02
Inline Method					**Replace @Test w/ assertThrowsassertThrows**				
Intercept	-4.437	0.148	**2e-16**	**0.016**	Intercept	-6.341	0.299	**2e-16**	**3e-04**
NOM	1.985	1.578	0.208	1.610	WMC	-3.140	5.816	0.589	**7e+02**
RFC	0.954	1.242	0.443	1.660	RFC	1.931	2.527	0.445	6e-04
AsD	-1.717	0.740	**0.020**	**0.201**	AsD	2.411	0.954	**0.011**	**1e+02**
Rename Class					**Replace try/catch with assertThrows**				
Intercept	-5.744	0.263	**2e-16**	**0.001**	Intercept	-7.364	0.442	**2e-16**	**3e-04**
LOC	4.972	2.048	**0.015**	**144.442**	WMC	-3.554	8.124	0.662	**7e+02**
NOM	-11.060	6.821	0.104	0.000	RFC	2.684	3.115	0.389	6e-04
AsD	0.813	0.979	0.406	2.255	AsD	3.464	1.251	**0.006**	**1e+02**
Replace Reserved Words					**Split conditional in statements**				
Intercept	-5.501	0.137	**2e-16**	0.001	Intercept	-5.489	0.190	**2e-16**	4e-03
LOC	5.778	1.341	**0.000**	**0.004**	LOC	6.145	1.132	**0.000**	**4e+02**
NOM	-8.593	3.426	**0.012**	**0.000**	NOM	-3.003	2.131	0.158	0.158
AsD	3.961	0.455	**2e-16**	**52.521**	AsD	1.669	0.669	**0.012**	**5.30**

5.3.1 Are test classes performed in classes with low quality?

Table 5.1 reports the results of the *Logistic Regression Model* for the refk refactoring in the set of refactoring operations considered in the study and the Qm_i quality metric in {LOC, NOM, WMC, RFC, AsD}. In addition, Table 5.2 reports the results of the *Logistic Regression Model* for the ref_k refactoring in the set of refactoring operations and the Ts_i test smell in {AR, DA, ECT, ET, GF, LT}. For each variable, the tables report the value of the estimate, the SE standard error, and the p-value indicating the statistical significance. The latter is formatted in **bold** to indicate whether a $p - value < 0.05$.

Looking at Table 5.1, we considered only three quality metrics in the models and discarded the LOC, NOM, RFC, and WMC metrics in pairs from the models due to multi-collinearity. The AsD metric was the only one with no multi-collinearity and was statistically significant for the *Add Assert Explanation* refactoring. The value of the estimate was positive (2.69), meaning an increase in the assertion density leads to an increase in the likelihood of the refactoring occurring in the test class. The AsD metric was statistically significant for other refactorings, with a positive value for the estimate to the *Replace NOT Operator* (2.80), *Replace @Test Expected with assertThrows* (2.41), *Replace Try/Catch With AssertThrows* (3.46), *Replace Reserved Words* (3.96), and *Split Conditional in statements* (1.66) refactorings. The RFC metric was also statistically significant, with a positive value for the *Extracted Method* (2.72) refactoring. The LOC metric was statistically significant with a positive value for the *Rename Class* (4.97), *Replace Reserved Words* (5.77), *Replace NOT operator* (6.34), and *Split Conditional in Statements* (6.14) refactorings.

After building the statistical models, we analyzed the Odds Ratio (OR) to measure the likelihood the independent variable has triggered a refactoring operation and the significance of their association. The last column of Table 5.1 shows the OR values and those in *bold* present $p - value < 0.05$. Odds Ratios greater than 1 suggest a positive association, while those less than 1 suggest a negative association. For the *Add Assert Explanation* refactoring, the AsD metric shows a statistically significant Odds Ratio of 14.80 with a p-value=1.34e-04. In this case, the odds ratio of the event associated with the AsD metric increases by 14.80 when the refactoring is applied. The intercept values are also included for each refactoring, indicating the baseline odds ratio of the dependent variable when no refactoring is applied.

> **Summary$_{3.1.1}$.** *Our results show structural metrics ($_{Qm_i-ref_k}$), specifically the AsD metric, significantly influence the likelihood of most test refactorings. In addition, the Add Assert Explanation refactoring stands out with a statistically significant Odds Ratio (14.80) for the AsD metric. Therefore, we rejected the null hypothesis $Hn1_{Qm_i-ref_k}$.*

Turning to Table 5.2, we considered all test smells in the models except the ET test smell for the *Rename Class* refactoring. No test smell was statistically significant for the *Add Assert Explanation*, *Extract Class*, *Replace @Rule With AssertThrows*, and *Inline Method* refactorings. As for the other refactorings, the DA test smell was statistically significant with a positive value of estimate for the *Replace NOT Operator* (6.34), *Extract Method* (2.33), *Rename Class* (6.46), and *Replace Try/Catch With AssertThrows* (11.81) refactorings. That test smell was also statistically significant with a negative estimate value for the *Replace @Test Expected with assertThrows* (-32.80) refactoring. The *Extract Method* (AR: 1.45, DA: 2.33, ECT: 1.68, ET: -6.17, GF: 1.73) and *Replace Reserved Words* (AR: 6.27, DA: 4.89, ECT: -11.89, LT: 6.5) refactorings were those with more statistically significant test smells in their models. All the test smells that have a statistical significance for the models also have a statistical significance for the Odds Ratio. For instance, the Odds Ratio for the AR test smell is 529.9, suggesting a substantial increase in the odds of the event associated with the response variable when the AR test smell increases by one unit.

> **Summary$_{3.1.2}$.** *For Add Assert Explanation, Extract Class, Replace @Rule With AssertThrows, and Inline Method, no test smell reached statistical significance. However, several test smells exhibit statistical significance in influencing test refactorings for the remaining test refactorings. For example, the DA holds significance for seven test refactorings. Therefore, we can reject the null hypothesis $Hn2_{Ts_i-ref_k}$*

5.3.2 What are the effects of test refactorings?

Table 5.3 reports the analysis of the impact of the ref_k refactoring in the set of refactoring operations considered in the study for the Qm_i in {LOC, NOM, WMC, RFC, AsD}. In addition, Table 5.4 reports the impact of the ref_k refactoring concerning the Ts_i test smells in {AR, DA, ECT, ET, GF, LT}. For each variable, the tables report the percentage of test classes with no changes in the values of the variables before and after applying the refactoring ($\equiv$), the percentage of test classes where the values of the variables decreased after applying the refactoring ($\downarrow$), and the percentage of test classes

Table 5.2: Results for the statistical model considering the test smells (Ts_i).

Variables	**Estimate**	**SE**	**Pr(>|z|)**	**OR**	**Variables**	**Estimate**	**SE**	**OR**	**OR**
	Add Assert Argument					**Replace NOT operator**			
Intercept	-5.358	0.139	**2e-16**	**0.005**	Intercept	-5.843	0.189	**2e-16**	**2e-03**
AR	0.259	2.630	0.921	1.296	AR	4.770	2.226	**0.031**	**1e+02**
DA	1.891	1.605	0.239	6.628	DA	5.511	1.546	**0.000**	**2e+02**
ECT	1.183	2.479	0.633	3.264	ECT	-19.011	10.606	0.0730	5e-09
ET	2.220	3.691	0.548	9.203	ET	6.238	12.557	0.619	5e+02
GF	0.570	2.370	0.810	1.769	GF	-20.037	17.786	0.259	1e-09
LT	-2.333	7.104	0.743	0.097	LT	-29.129	29.848	0.329	2e-13
	Extract Class					**Replace conditional by Parameterized**			
Intercept	-4.383	0.092	**2e-16**	**0.012**	Intercept	-4.879	0.221	**2e-16**	**7e-03**
AR	-1.214	2.320	0.601	0.297	AR	4.175	14.587	0.774	6e+01
DA	-0.132	1.635	0.935	0.876	DA	-2e+01	2e+01	0.294	1e-11
ECT	1.524	1.467	0.299	4.591	ECT	-1e+03	6e+02	**0.018**	**0.000**
ET	0.712	1.729	0.680	2.038	ET	-9e+01	8e+1	0.226	3e-43
GF	-5.470	4.570	0.231	0.004	GF	-8e+01	1e+02	0.483	1e-38
LT	2.454	2.157	0.255	1e+01	LT	-58.729	1e+02	0.627	3e-26
	Extract Method					**Replace Rule w/ assertThrows**			
Intercept	-2.959	0.045	**2e-16**	**0.052**	Intercept	-6.106	0.209	**2e-16**	**2.2e-03**
AR	1.459	0.703	**0.038**	**4.304**	AR	3.516	2.284	0.124	3.4e+01
DA	2.330	0.483	**0.000**	**10.281**	DA	-3.613	4.026	0.370	2.7e-02
ECT	1.687	0.737	**0.022**	**5.402**	ECT	-1.932	5.394	0.720	1.4e-01
ET	-6.176	1.783	**0.001**	**0.002**	ET	2.838	3.192	0.374	1.7e+01
GF	1.739	0.635	**0.006**	**5.691**	GF	-2.638	7.520	0.726	7.2e-02
LT	2.089	1.490	0.161	8.078	LT	0.429	7.752	0.956	1.5e+00
	Inline Method					**Replace @Test w/ assertThrows**			
Intercept	-4.687	0.101	**2e-16**	**0.009**	Intercept	-5.477	0.180	**2e-16**	**4.2e-03**
AR	1.036	1.761	0.556	2.819	AR	0.865	3.830	0.821	2.4e+00
DA	0.969	1.256	0.440	2.636	DA	-32.840	15.920	**0.039**	**5.5e-15**
ECT	0.541	2.108	0.797	1.718	ECT	-2.712	7.397	0.714	6.6e-02
ET	0.170	2.312	0.941	1.185	ET	0.497	3.064	0.871	1.6e+00
GF	-0.662	2.259	0.769	0.516	GF	-2e+03	1e+05	0.984	0.000
LT	1.401	3.020	0.643	4.061	LT	5.726	6.188	0.355	3.1e+02
	Rename Class					**Replace try/catch w/ assertThrows**			
Intercept	-5.550	0.170	**2e-16**	**3e-03**	Intercept	-6.524	0.276	**2e-16**	**1.5e-03**
AR	1.365	4.339	0.753	3.920	AR	-29.561	11.610	**0.011**	**1e-13**
DA	6.448	1.832	**0.000**	**6e+02**	DA	11.811	3.175	**0.000**	**1e+05**
ECT	-33.873	15.842	**0.032**	**1e-15**	ECT	-14.973	16.355	0.360	3e-07
GF	-9.121	8.968	0.309	1e-04	ET	6.716	3.322	**0.043**	**8e+02**
LT	-6.716	6.591	0.308	1e-03	GF	5.124	7.030	0.466	1e+02
					LT	1.338	7.303	0.855	3.8e+00
	Replace Reserved Words					**Split conditional in statements**			
Intercept	-4.506	0.100	**2e-16**	**1e-02**	Intercept	-5.128	0.120	**2e-16**	**4e-03**
AR	3.844	1.56	**0.014**	**4e+01**	AR	2.892	1.743	0.014	**4e+01**
DA	4.575	1.077	**2e-05**	**9e+01**	DA	4.374	0.878	**6e-07**	**1e+01**
ECT	-20.498	6.986	**0.003**	**1e-09**	ECT	-3.389	3.684	0.357	9e-06
ET	-9.720	3.640	**0.007**	**6e-05**	ET	1.197	3.094	0.698	1e-02
GF	-20.910	10.306	**0.042**	**8e-10**	GF	-2.340	2.770	0.398	7e-05
LT	6.074	1.632	**0.000**	**4e+02**	LT	-1.571	4.857	0.746	2e-06

where the values of the variables increased after applying the refactoring (↑). In addition, we performed the *Wilcoxon Rank Sum Test* to analyze whether there is a statistically

significant difference between the distributions corresponding to the test code quality before and after refactorings, with the $p - value$ indicating the statistical significance.

Table 5.3: Results of decrease and increase of quality and Wilcoxon Rank Sum Test for Metrics (Qm_i)

Metrics	≡ (%)	↓ (%)	↑ (%)	p-value	Metrics	≡ (%)	↓ (%)	↑(%)	p-value
Add Assert Argument					**Replace NOT Operator**				
LOC	0.00	28.57	71.43	0.90	LOC	54.55	9.09	36.36	1.00
NOM	16.67	16.67	66.67	0.80	NOM	90.91	9.09	0.00	0.97
WMC	14.29	28.57	57.14	0.85	WMC	90.91	9.09	0.00	1.00
RFC	0.00	28.57	71.43	0.80	RFC	45.45	18.18	36.36	0.92
AsD	28.57	28.57	42.86	1.00	AsD	81.82	9.09	9.09	1.00
Extract Class					**Replace conditional by Parameterized**				
LOC	60.87	26.09	13.04	0.90	LOC	9.09	81.82	9.09	0.67
NOM	56.52	34.78	8.70	0.80	NOM	72.73	27.27	0.00	0.55
WMC	60.87	30.43	8.70	0.85	WMC	63.64	27.27	9.09	0.67
RFC	56.52	34.78	8.70	0.80	RFC	27.27	63.64	9.09	0.58
AsD	39.13	34.78	26.09	1.00	AsD	45.45	18.18	36.36	0.95
Extract Method					**Replace Rule w/ assertThrows**				
LOC	61.70	21.28	17.02	0.43	LOC	0.00	100.00	0.00	0.013
NOM	58.51	30.85	10.64	0.25	NOM	100.00	0.00	0.00	NaN
WMC	62.77	25.53	11.70	0.29	WMC	100.00	0.00	0.00	NaN
RFC	60.64	21.28	18.09	0.37	RFC	0.00	0.00	100.00	0.013
AsD	38.30	31.91	29.79	0.93	AsD	0.00	0.00	100.00	0.013
Inline Method					**Replace @Test w/ assertThrows**				
LOC	31.25	43.75	25.00	0.87	LOC	0.00	88.89	11.11	0.42
NOM	31.25	37.50	31.25	0.62	NOM	88.89	11.11	0.00	1.00
WMC	31.25	37.50	31.25	0.81	WMC	55.56	44.44	0.00	0.74
RFC	25.00	37.50	37.50	0.92	RFC	0.00	88.89	11.11	0.25
AsD	56.25	18.75	25.00	0.82	AsD	11.11	0.00	88.89	0.10
Rename class					**Replace Try/Catch w/ AssertThrows**				
LOC	50.00	0.00	50.00	0.08	LOC	0.00	0.00	0.00	NaN
NOM	50.00	0.00	50.00	0.09	NOM	0.00	0.00	0.00	NaN
WMC	50.00	0.00	50.00	0.09	WMC	0.00	0.00	0.00	NaN
RFC	50.00	0.00	50.00	0.12	RFC	0.00	0.00	0.00	NaN
AsD	50.00	50.00	0.00	1.00	AsD	0.00	0.00	0.00	NaN
Replace Reserved Words					**Split conditional in statements**				
LOC	64.29	11.90	23.81	0.98	LOC	65.22	0.00	34.78	0.89
NOM	76.19	9.52	14.29	0.91	NOM	91.30	0.00	8.70	0.96
WMC	71.43	11.90	16.67	0.91	WMC	86.96	0.00	13.04	0.97
RFC	64.29	11.90	23.81	0.91	RFC	56.52	13.04	30.43	0.96
AsD	71.43	14.29	14.29	0.86	AsD	91.30	4.35	4.35	0.96

When analyzing the values of the Qm_i metric in Table 5.3, we noticed an increase in the metrics values of most test classes after applying the *Add Assert Argument* and *Rename Class* refactorings. For example, 71.43% of test classes where developers applied the *Add Assert Argument* refactoring increased the LOC metric, 66.67% increased the NOM metric, 57.14% increased the WMC metric, 71.43% increased the RFC metric, and 42.86% increased the AsD metric. Thus, evidence exists that the main goal of those refac-

torings might not be associated with improving the size, coupling, and complexity of test codes. While most test classes kept the same values for quality metrics after applying the *Extract Class*, *Extract Method*, and *Inline Method* refactorings, at least other 30% showed a decrease in metrics values after applying those refactorings. We expect those results as refactorings are from Fowler's catalog and are known for improving code organization and redistributing responsibilities. With concerns to test-specific refactorings, at least 63% of the test classes showed an improvement of the LOC metric after applying the *Replace Rule w/ assertThrows*, *Replace Conditional by ParameterizedTest*, and *Replace @Test with assertThrows* refactorings. In addition, the last two refactorings also showed an improvement in the RFC and WMC metrics.

In addition, Table 5.3 shows a statistically significant difference in three metrics (LOC: 0.01, RFC: 0.01, and AsD: 0.01) before and after the *Replace Rule with assertThrows* refactoring. For the remaining refactorings, no statistically significant differences exist. In addition, the *Wilcoxon Rank Sum Test* did not perform for some refactoring types (e.g., the *Replace @Rule with assertThrows*, and *Replace try/catch with assertThrows* refactorings), indicated by NaN values; maybe due to the absence of data and the difference in the *Replace Rule with assertThrows* refactoring.

> ❯ **Summary$_{3.2.1}$.** *Several structural metrics did not exhibit statistical significance differences before and after applying the test refactorings; therefore, accept the null hypothesis* $Hn3_{ref_k-Qm_i}$ *for most of the refactorings. Yet, the test code quality had a slight improvement in terms of complexity, coupling, and size when considering the refactorings from Fowler's catalog and some test-specific refactorings (i.e., the Replace Rule w/ assertThrows, Replace Conditional by ParameterizedTest, and Replace @Test with assertThrows refactorings).*

Focusing on Table 5.4, we observe most test classes where developers applied refactorings from Fowler's catalog retain the same values of test smells. However, around 30% of the test classes presented an improvement in terms of the AR and ECT test smells after the *Extract Class* and *Extract Method* refactorings. Interestingly, around 21% of the test classes had an improvement in terms of the LT and DA test smells with the *Extract Class* and the GF test smell with the *Extract Method* refactorings. The *Replace Conditional by Parameterized* and *Replace @Test w/ assertThrows* test-specific refactorings decreased the test smells in around 30% of the test classes. Other test-specific refactorings, such as the *Replace Reserved Words*, *Replace NOT operator* and *Split Conditional into Statements* refactorings, did not contribute as much to the decrease of test smells

Table 5.4: Results of decrease and increase of quality and Wilcoxon Rank Sum Test for Metrics (Ts_i)

Metrics	≡ (%)	↓ (%)	↑ (%)	p-value	Metrics	≡ (%)	↓ (%)	↑ (%)	p-value
Add Assert Argument					**Replace NOT Operator**				
AR	57.14	14.29	28.57	1.00	AR	81.82	18.18	0.00	0.97
ECT	42.86	28.57	28.57	0.95	ECT	81.82	18.18	0.00	0.65
GF	80.00	0.00	20.00	0.85	GF	100.00	0.00	0.00	0.53
ET	85.71	0.00	14.29	1.00	ET	81.82	18.18	0.00	0.17
LT	85.71	0.00	14.29	0.77	LT	81.82	18.18	0.00	NaN
DA	57.14	14.29	28.57	1.00	DA	90.91	9.09	0.00	0.17
Extract Class					**Replace Conditional by Parameterized**				
AR	65.22	30.43	4.35	1.00	AR	45.45	54.55	0.00	0.01
ECT	47.83	43.48	8.70	0.95	ECT	63.64	36.36	0.00	NaN
GF	86.96	8.70	4.35	0.85	GF	63.64	36.36	0.00	0.04
ET	86.96	8.70	4.35	1.00	ET	90.91	9.09	0.00	0.36
LT	69.57	21.74	8.70	0.77	LT	81.82	18.18	0.00	0.04
DA	73.91	21.74	4.35	1.00	DA	100.00	0.00	0.00	0.17
Extract Method					**Replace Rule w/ AssertThrows**				
AR	59.57	29.79	10.64	0.92	AR	0.00	0.00	100.00	0.01
ECT	59.57	30.85	9.57	0.94	ECT	100.00	0.00	0.00	NaN
GF	74.47	21.28	4.26	0.63	GF	100.00	0.00	0.00	NaN
ET	88.30	6.38	5.32	0.87	ET	100.00	0.00	0.00	NaN
LT	87.23	6.38	6.38	0.65	LT	100.00	0.00	0.00	NaN
DA	78.72	13.83	7.45	0.90	DA	100.00	0.00	0.00	NaN
Inline Method					**Replace @Test w/ assertThrows**				
AR	75.00	0.00	25.00	0.82	AR	22.22	33.33	44.44	1.00
ECT	62.50	0.00	37.50	0.87	ECT	55.56	44.44	0.00	0.37
GF	75.00	6.25	18.75	0.74	GF	55.56	44.44	0.00	0.10
ET	93.75	0.00	6.25	0.98	ET	77.78	22.22	0.00	0.25
LT	93.75	0.00	6.25	0.71	LT	44.44	55.56	0.00	0.03
DA	87.50	0.00	12.50	0.97	DA	88.89	11.11	0.00	0.01
Rename Class					**Replace Try/Catch w/ AssertThrows**				
AR	100.00	0.00	0.00	0.42	AR	0.00	0.00	0.00	NaN
ECT	100.00	0.00	0.00	0.66	ECT	0.00	0.00	0.00	NaN
GF	100.00	0.00	0.00	0.51	GF	0.00	0.00	0.00	NaN
ET	100.00	0.00	0.00	0.82	ET	0.00	0.00	0.00	NaN
LT	100.00	0.00	0.00	0.97	LT	0.00	0.00	0.00	NaN
DA	100.00	0.00	0.00	0.66	DA	0.00	0.00	0.00	NaN
Replace Reserved Words					**Split Conditional into Statements**				
AR	83.33	4.76	11.90	0.93	AR	91.30	4.35	4.35	0.91
ECT	83.33	7.14	9.52	0.89	ECT	86.96	4.35	8.70	0.70
GF	97.62	2.38	0.00	0.67	GF	100.00	0.00	0.00	0.97
ET	90.48	9.52	0.00	0.40	ET	95.65	4.35	0.00	0.34
LT	88.10	9.52	2.38	0.33	LT	95.65	4.35	0.00	NaN
DA	92.86	2.38	4.76	0.44	DA	95.65	4.35	0.00	0.34

maybe because they are intrinsically related to the InA test smell, not supported by the detection tools.

Regarding the *Wilcoxon Rank Sum Test*, Table 5.4 shows statistically significant differences in test smells before and after three refactorings. The AR metric differs among

versions when applying the *Replace Conditional by ParameterizedTest* and *Replace Rule with assertThrows* refactorings. Besides the AR metric, ECT and GF metrics differed among the versions when applying the *Replace Conditional by ParameterizedTest* refactoring. The DA and LT test smells also differed among the versions when applying the *Replace @Test with assertThrows* refactoring. Like Table 5.3, the *Wilcoxon Rank Sum Test* did not perform for some refactorings.

> **Summary$_{3.2.2}$.** *The Replace @Rule with assertThrows, Replace @Test with assert-Throws and Replace conditional bt ParameterizedTest refactorings influenced the GF, LT, AR, and ECT test smells. As most test refactorings did not differ statistically significantly, we accepted the* $Hn4_{ref_k - Qm_i}$ *null hypothesis.*

5.4 DISCUSSION

When analyzing whether the low-quality test code drives test refactorings, we observed the AsD metric is related to most test refactorings, indicating the more assertions in the test code, the more likely developers would refactor it. However, test refactorings, such as the *Rename Class*, *Replace @Rule with assertThrows*, and *Replace conditional by ParameterizedTest* refactorings, were not related to the number of assertions because they refer to changes in the test structure or address some other non-density related concern. For example, developers usually place assertions within loop structures in the test code to assert the same condition with different values. With the *Replace conditional by ParameterizedTest* refactoring, developers remove the loop structure, but the test method continues under tests with a set of values passed as parameters. As for the negative relation of the *Extract Class* and *Extract Method* refactorings to the AsD metric, we can understand it allows classes and methods to focus on specific aspects of the test logic, leading to a more modular structure. For example, some test methods implement hard logic to stimulate the production class but have few assertions, indicating developers could place logic into another class or method to allow reusability.

In addition, unexpected results when analyzing whether the presence of test smells drives the test refactorings. The *Add Assert Argument* refactoring is applied to solve the AR test smell, and the *Replace try/catch with assertThrows* refactoring is often used to remove the ECT test smell. However, no significant correlation exists between them, indicating those refactorings can stem from variations in coding styles or project-specific guidelines. As a result, the *Replace NOT Operator*, *Replace Reserved Words*, and *Split*

conditional in statements refactorings positively relate to the AR test smell. It is explainable as most assertions in the test code do not have an explanatory message, resulting in an AR, test smell, and those refactorings occur within the assertions to change their parameters. Meanwhile, it is also interesting to notice that while the *Extract Method* refactoring is positively related to most test smells, it is negatively related to or does not have a relationship with the test smells responsible for indicating the method has spread or tangled concerns, i.e., the ET and LT test smells. It might indicate developers use the *Extract Method* refactoring for general code improvement but not to solve those specific test smells.

Finally, the analysis of whether the test refactorings improve the code quality shows that most test classes maintained stable metric values. Yet, approximately 30% of test classes exhibited a decrease in metrics after applying refactorings from Fowler's catalog. Despite the overall stability, some test classes experienced enhancements, particularly in test smells such as the AR and ECT test smells. While improving the test code in terms of quality metrics is expected after applying those refactorings, they can yield substantial benefits concerning specific test smells. Concerning the test-specific refactorings, notably, the *Replace Conditional by ParameterizedTest* and *Replace @Test w/ assertThrows* refactorings demonstrated positive effects, contributing to a significant reduction in test smells and quality metrics in approximately 30% of the test classes. Although we could notice test refactorings influence the test code quality regarding structural metrics and test smells, we could not observe a statistical significance. The lack of statistical significance could be due to a limited number of instances of the test refactorings in the dataset.

5.5 THREATS TO VALIDITY

This section discusses the potential threats that can affect the validity of our study.

Construct validity. The first threat concerns the test smells and structural metrics used to assess the test code quality. We did not calculate all the Chidamber & Kemerer metrics as some do not apply to the context of the test code (e.g., *Depth Inheritance Tree* - DIT). Nevertheless, we chose a mix of metrics capturing the test code size, coupling, and cohesion. Another threat to validity concerns the identification of test smells and refactoring operations. We used tools validated and used by the research community. Although the tools present high precision and recall scores, they can report

some false positive or false negative instances of test smells or refactorings. In response to this limitation, we performed a preliminary and manual investigation to assess the degree of accuracy of the tools before running them on a large scale.

Internal Validity. That category of threats to validity concerns by-product changes of other maintenance activities (e.g., bug fixes or changes in requirements) that could also contribute to removing test smells. Therefore, the data analysis does not indicate a causal relationship, but there is a possibility of a relationship to investigate in the future.

External Validity. That class of threats to validity mainly concerns the subject projects of our study. We selected open-source JAVA projects from GITHUB, which are only a fraction of the complete picture of open-source software and do not necessarily represent industrial practices. Therefore, we can not generalize the results to the industrial context and other programming languages.

Conclusion validity. To address how frequently test refactoring on test classes affects quality concerns, we used logistic regression models to identify correlations. Other than highlighting cases of significant correlations, we reported and discussed the Odds Ratio values. In addition, to investigate the effect of test refactoring on test code quality, we also employed well-established statistical tests such as the Wilcoxon Rank Sum Test (MCKNIGHT; NAJAB, 2010).

5.6 CHAPTER SUMMARY

This chapter presented a study to understand whether the test code quality indicates which test classes are more likely to be refactored and to what extent test refactoring operations can improve the test code quality. We conducted that study on a set of open-source JAVA projects, starting from collecting data on the test code quality, test smells, and refactoring operations arising in the major releases of the projects. Then, we employed statistical approaches to address the goals of our investigation.

The findings from that study can benefit researchers and practitioners from multiple perspectives. In the first place, our research can reveal insights into the refactoring types that can deteriorate test code quality. Such information would be relevant for researchers in both the fields of refactoring and testing, as it can lead them to (1) extend the knowledge of the best and bad practices to apply test refactoring properly; (2) devise

novel test refactoring approaches aware of the possible side effects of refactoring, e.g., we can envision multi-objective search-based refactoring approaches to optimize refactoring recommendations based on both quality attributes; and (3) design novel recommendation systems to support developers in understanding how refactorings can impact different test code properties. The results would also be useful to practitioners, who can have additional proof of the side effects of refactoring, hence possibly being stimulated further on the need to employ automated refactoring tools. In the second place, our findings indicate the nature of the test cases more likely to be subject to refactoring operations. Researchers can use that information to define refactoring recommenders and refactoring prioritization approaches, while practitioners can acquire awareness of their actions.

In summary, this chapter provided with the following key contributions:

1. An empirical understanding of the factors triggering test refactoring operations, which comprises an analysis of how test code quality comes into play;

2. Evidence of the impact of test refactoring on test code quality;

3. An online appendix (MARTINS *et al.*, 2023d) which provides all material and scripts employed to address the goals of the study.

Chapter 6

DEVELOPER-ORIENTED TEST REFACTORING RECOMMENDATIONS

While academic research recommends refactoring to fix test smells and improve structural metrics (LAMBIASE *et al.*, 2020; PECORELLI *et al.*, 2020b; SANTANA *et al.*, 2020), a gap exists that consists in the lack of adoption of refactoring tools in practice (KIM; ZIMMERMANN; NAGAPPAN, 2014). That gap is attributed, in part, to the need for more support for the types of problems developers face when refactoring and the high number of false positives.

Recent studies have explored two approaches to bridge the gap in understanding test refactorings: i) Analyzing Stack Exchange sites to comprehend the motivations behind refactorings (PERUMA *et al.*, 2022; MARTINS *et al.*, 2023); and ii) Submitting pull requests to open-source projects to gain insights into the developer's perspective on test refactorings, particularly those aimed at addressing test smells (SOARES *et al.*, 2022). These studies provide valuable insights into the motivations and perspectives of developers regarding test refactorings, helping to advance our understanding of this critical aspect of software development.

Machine Learning (ML) emerges as a promising solution to overcome the need for more tool adoption in software development. ML algorithms can support identifying refactoring opportunities and recommending proper refactorings that align closely with developers' practices (SAGAR *et al.*, 2021; ANICHE *et al.*, 2022). For instance, Aniche *et al.* (2022) investigated the effectiveness of six supervised ML algorithms trained on

code smells and structural metrics to predict 20 software refactorings from Fowler's catalog. While these algorithms performed well in predicting refactoring opportunities in production code, it's important to note that their findings may not be directly applicable to test code.

In this chapter, we present our investigation of the performance of supervised ML algorithms in classifying the developers' intention to apply test refactoring, answering **RQ_3: How accurately can we suggest test refactoring operations for fixing test smells using ML techniques?**. That investigation involved two distinct classification tasks: (i) the classification of code changes where developers would perform some test refactoring and (ii) the classification of specific test refactoring operations developers apply. To conduct it, we built a dataset encompassing information on 50 different test refactoring operations on ten open-source JAVA projects. In addition, the dataset comprises 21 test smells identified in test classes, 5 test metrics extracted from test classes, and 13 process metrics extracted from both production and test classes. Then, we trained six ML algorithms to select the best one for classifying the developers' intentions and the specific test refactoring operations.

The remaining chapter is structured as follows. Section 6.2 outlines the context of our study and the methods used to address the research questions. Section 6.3 reports on the results of our work. Section 6.4 discusses additional insights and implications of our results. Section 6.5 outlines the potential threats to validity.

6.1 RESEARCH QUESTIONS AND OBJECTIVES

Our *goal* was to investigate the effectiveness of supervised ML algorithms in classifying the code changes where practitioners apply test refactoring actions, considering two levels of granularity: i) the classification of code changes where practitioners apply a refactoring in test classes and ii) the classification of the specific test refactoring action applied by practitioners in a certain code change. Both levels have the *purpose* of studying the feasibility of an automated instrument for test code quality assurance, which practitioners can use to identify test classes requiring maintenance effort and assess how to improve them. The *perspective* is of researchers and practitioners: the former are interested in understanding the performance and limitations of classification models to identify test refactoring opportunities. At the same time, the latter assesses how feasible the proposed automated solution of test code quality assurance would be in practice.

Our empirical investigation slued round two research objectives. As a first step, we explored how six supervised ML algorithms (Decision Trees - DT, Naive Bayes - NB, Logistic Regression - LR, Extra-Tree - ExT, Support Vector Machine - SVM, and Random Forest - RF) accurately classify the code changes where developers would apply test refactoring. Then, we verified the performance of classifying a specific test code refactoring operation. So, we asked:

RQ$_{3.1}$. *How accurate are supervised ML algorithms in classifying the code changes where developers would do test refactoring?*

After analyzing the performance of ML models in classifying the developer's intention to perform test refactoring, we verified the performance of classifying a specific test code refactoring operation. So, we asked:

RQ$_{3.2}$. *How accurate are supervised ML algorithms in classifying specific test refactoring operations?*

6.2 EXPERIMENTAL DESIGN

This section reports the research method we applied to address our **RQs**.

6.2.1 Context of the Study

The context of our study consisted of ten open-source Java projects that met the following selection criteria: i) we focused on open-source projects, as we needed access to change history information; ii) we decided to rely on popular and real-world projects having enough releases to collect data; and iii) we standardized the building process to streamline build configurations across all projects and use supplementary tools (i.e., the TestRefactoringMiner and VITRuM tools). As such, we used the SEART tool[1] to select open-source and non-fork projects from GitHub with at least 100 stars, 10 major releases, 1,000 lines of code, and 5 test classes. In addition, we sought Java projects for compiling with Maven and Java 8—Java 8 is the most popular Java version used nowadays[2].

Table 6.1 reports, for each project, the number of Lines of Code (LOC), Test Meth-

[1] <https://seart-ghs.si.usi.ch/>
[2] <https://www.jetbrains.com/lp/devecosystem-2021/java/>

ods (NOM), Test Refactorings, and Test Classes. Those projects together have a history of 1,270 tags and 120,107 commits measured at the moment of data collection in October 2023.

Table 6.1: Overview of the selected JAVA open-source projects.

Owner / Repository name	LOC	NOM	# Test Classes	# Refactored Classes (RQ1)	# Refactorings per Type (RQ2)
graphhopper/graphhopper	54,654	3,870	182	96	154
itext/itext7	102,215	8,189	332	250	318
fabric8io/kubernetes-client	5,946	776	78	33	42
apache/iotdb	4,371	250	23	9	9
nationalsecurityagency/emissary	26,296	2,570	239	152	295
seleniumhq/htmlunit-driver	1,420	201	8	4	6
cmu-phil/tetrad	304	30	10	4	5
questdb/questdb	190,559	37,465	562	258	424
zanata/zanata-platform	2,923	229	17	12	17
googleapis/google-http-java-client	2,799	269	20	8	13
Total	**391,417**	**53,849**	**1,471**	**826**	**1,283**

6.2.2 Dependent Variables

The dependent variable of our study is a binary variable that shows the presence or absence of test refactoring operations within the code changes of the projects. That variable operationalizes differently depending on the RQ, as explained in the following.

We used the TestRefactoringMiner tool (MARTINS *et al.*, 2023a) to analyze the test code changes from the oldest ones to the most recent revision. Building of that tool was on top of the RefactoringMiner state-of-the-art tool, which has the highest accuracy among the currently available refactoring mining tools, with an average precision of 99.8% and recall of 97.6% (TSANTALIS; KETKAR; DIG, 2020). In particular, the TestRefactoringMiner tool analyzes the added, deleted, and changed files between two project versions to detect specific test refactorings.

Fowlers' catalog (FOWLER, 1999) presents a set of refactorings commonly applied to test and production code. Hence, we considered refactorings from his catalog: 1) Add Class Annotation, 2) Add Method Annotation, 3) Encapsulate Attribute, 4) Extract And Move Method, 5) Extract Attribute, 6) Extract Class, 7) Extract Interface, 8) Extract Method, 9) Extract Subclass, 10) Extract Superclass, 11) Extract Variable, 12) Inline Attribute, 13) Inline Method, 14) Inline Variable, 15) Invert Condition, 16) Merge Attribute, 17) Merge Class, 18) Merge Method, 19) Merge Variable, 20) Modify Class Annotation,

21) Modify Method Annotation 22) Move And Inline Method, 23) Move Attribute, 24) Move Class, 25) Move Method, 26) Pull Up Attribute, 27) Pull Up Method, 28) Push Down Attribute, 29) Push Down Method, 30) Remove Class Annotation, 31) Remove Method Annotation, 32) Replace Anonymous With Lambda, 33) Replace Attribute, 34) Replace Attribute With Variable, 35) Replace Loop With Pipeline, 36) Replace Pipeline With Loop, 37) Replace Variable With Attribute, 38) Split Attribute, 39) Split Class, 40) Split Conditional, and 41) Split Variable. In addition, we considered other test-specific refactorings applied only to test code: 42) Add Assert Argument, 43) Replace @test(expected) with assertThrows, 44) Replace Conditional by ParameterizedTest, 45) Replace NOT operator, 46) Replace Reserved Words, 47) Replace Rule With AssertThrows, 48) Replace Try/Catch With AssertThrows, 49) Replace Try/Catch With Rule, and 50) Split Conditional Statement in Assertions. A complete description of test refactorings is available in our online appendix (MARTINS *et al.*, 2023e). All the refactorings are collected based on what developers modified throughout the projects' lifecycle; therefore, they indicate the code changes where developers applied test refactoring. In the context of $\mathbf{RQ}_{3.1}$, we analyzed the performance in classifying those test code changes. As for $\mathbf{RQ}_{3.2}$, we independently considered the various refactoring operations and analyzed the performance in classifying specific test refactoring.

6.2.3 Independent Variables

To verify the extent to which statically computable metrics we can adopt to classify test code refactorings, we considered 39 features along three dimensions: test code metrics, test smells, and process metrics. Those features are the independent variables of the study, and we chose them based on previous researches that show their impact on quality aspects of test code (SPADINI *et al.*, 2018; CATOLINO *et al.*, 2019; PECORELLI; PALOMBA; LUCIA, 2021; KIM; CHEN; YANG, 2021). For space limitations, a description of the independent variables is available in our online appendix (MARTINS *et al.*, 2023e).

Test Code Metrics. Features in this category derive from the test code attributes. In particular, we selected six test code metrics related to size, complexity, and coupling. Although many automated tools calculate structural metrics of source codes (e.g., the Eclipse Metrics and CK Metrics tools), we used the VITRuM plug-in specifically designed to calculate and present the visualization of test-related metrics (PECORELLI *et al.*, 2020b). More specifically, we collected the metrics: 1) Lines of Code (LOC), 2)

Number of Methods (NOM), 3) Weighted Method per Class (WMC), 4) Response for a Class (RFC), and 5) Assertion Density (AD).

Test Smells. Test smells are bad design or implementation choices in the test code (DEURSEN *et al.*, 2001). In particular, we considered 21 test smells detected by the TSDETECT tool (PERUMA *et al.*, 2020a). Among the test smell detection tools available for JAVA code (ALJEDAANI *et al.*, 2021), that tool presents the highest accuracy, with an average precision score of 96% and an average recall score of 97%. It reports either (i) the presence or absence of a test smell in a test class or (ii) the number of instances per test smell in a test class. In addition, it receives a configuration of the severity thresholds for each test smell (SPADINI *et al.*, 2020). We ran it with default values for the severity thresholds and detected the test smells: 1) Assertion Roulette (AR), 2) Constructor Initialization (CI), 3) Conditional Test Logic (CTL), 4) Default Test (DT), 5) Dependent Test (DpT), 6) Duplicate Assert (DA), 7) Eager Test (ET), 8) Exception Handling (ECT), 9) Empty Test (EpT), 10) General Fixture (GF), 11) Ignored Test (IgT), 12) Lazy Test (LT), 13) Mystery Guest (MG), 14) Magic Number Test (MNT), 15) Redundant Assertion (RA), 16) Resource Optimism (RO), 17) Redundant Print (RP), 18) Verbose Test (VT), 19) Sensitive Equality (SE), 20) Sleepy Test (ST), and 21) Unknown Test (UT).

Process Metrics. Since refactoring represents an activity based on the developer's experience, we computed and analyzed several ad-hoc metrics extracted from Git repositories. Specifically, we ran the PYDRILLER tool (SPADINI; ANICHE; BACCHELLI, 2018) to collect: 1) Code Churn Max (CCM), 2) Code Churn Average (CCA), 3) Commits Count (Co), 4) Contributors Count (Con), 5) Minor Contributors Count (MCon), 6) Contributors Experience (ConE), 7) Lines Added Count (ALC), 8) Lines Added Max (ALM), 9) Lines Added Average (ALA), 10) Lines Removed Count (RLC), 11) Lines Removed Max (RLM), and 12) Lines Removed Average (RLA).

6.2.4 Research Method

This section discusses the research methods employed to address our RQs. It is important to emphasize that the research method performed was the same for both **RQ**s, while the level of analysis was different. In $\mathbf{RQ}_{3.1}$**: How accurate are supervised ML**

algorithms in classifying the code changes where developers would do test refactoring?, we classified the code changes where practitioners would do some test refactoring. In **$RQ_{3.2}$: How accurate are supervised ML algorithms in classifying specific test refactoring operations?**, we classified specific test refactoring operations performed by practitioners.

The first step is related to the feature selection to identify the relevant metrics to use as predictors. We quantified the predictive power of each metric in terms of information gain (QUINLAN, 1986) to measure how much a model would benefit from the presence of a metric. At the end, we considered the metrics having an information gain higher than zero as predictors, i.e., we discarded the metrics that did not provide any beneficial effect.

After completing the feature selection, we identified the best ML algorithm. The literature on test refactoring classification is embryonic; therefore, we took this opportunity to benchmark learning algorithms with different characteristics. We evaluated the Logistic Regression (LR) (LAVALLEY, 2008), Naive Bayes (NB) (KHOMH *et al.*, 2009), (NOBLE, 2006), Decision Tree (FREUND; MASON, 1999), Random Forest (RF) (HO, 1995), and Extra-Tree (GEURTS; ERNST; WEHENKEL, 2006) algorithms. To assess the performance of our models, we performed a walk-forward validation (FALESSI *et al.*, 2020), applying it to individual projects. We decided to apply that validation because we relied on temporal data, so it is crucial to maintain the chronological order to avoid data leakage. In the k-fold cross-validation (widely used), the data is randomly partitioned, which can break the temporal structure.

In a walk-forward validation, the dataset reports a time series we can divide into chronologically orderable parts, e.g., a project release or a commit. In each run, all data available before the part to predict are used as the training set, while the part to predict is used as the test set, preventing the test set from having data antecedent to the training set. Specifically, the number of iterations equals the number of parts minus one. We trained each model on the first n releases and tested on the $(n+1)$-th release. After splitting the training and test sets, we normalized the metric values through the *min-max scaling* to perform a realistic validation of the model where the training and test sets were individually normalized based on their distributions. To implement the algorithms, we employed the SCIKIT-LEARN library (KRAMER, 2016) in PYTHON, which provides public APIs to configure, execute, and validate all the above-mentioned classifiers.

After collecting the performance of the algorithms, we statistically verified our con-

clusions by using the Friedman (SHELDON; FILLYAW; THOMPSON, 1996) and Nemenyi (NEMENYI, 1963) tests on the distribution of F-Measure values of ML models over the different projects and test refactoring operations. We used the former to determine whether or not there is a statistically significant difference between the F-Measure values and the latter to report its results using MCM (i.e., Multiple Comparisons with the best) plots (MCMINN, 2004). We used 0.05 as a significance level, so the elements plotted above the gray band were statistically larger than the others. In addition, the dots in the plot represent the median MCC of the algorithms obtained in the projects: a blue dot indicates the MCC of an algorithm was statistically better than the other algorithms. In contrast, red dots indicate the performance was not statistically different. At the end of this step, we selected the best model to classify the dependent variable.

It is important to point out that test refactoring is an unbalanced problem. The number of test cases refactored instances represents almost 2% of the total amount of test cases on our dataset. As such, the problem was largely underrepresented, threatening the ability of ML algorithms to learn the characteristics of test refactoring properly. For this reason, we tried to improve the performance by experimenting with several under- and over-sampling techniques to balance the data. As for under-sampling, we experimented the advanced NearMiss 1, NearMiss 2, and NearMiss 3 algorithms (YEN; LEE, 2006). We also experimented with the Random Undersampling approach, which randomly explores the distribution of majority instances and under-samples them. In terms of over-sampling approaches, we experimented with SMOTE (CHAWLA *et al.*, 2002), and advanced versions of this algorithm such as ADASYN (HE *et al.*, 2008) and Borderline-SMOTE (HAN; WANG; MAO, 2005). In addition, we also experimented with the Random Oversampling approach, which randomly explores the distribution of the minority class and over-samples them.

Once we have collected the performance of the various algorithms, we statistically verified our conclusions by using the Friedman (SHELDON; FILLYAW; THOMPSON, 1996) and Nemenyi (NEMENYI, 1963) tests on the distribution of F-Measure values of the ML models over the different projects and test refactoring operations. We used the former to determine whether or not there is a statistically significant difference between the F-Measure values and the latter to report its results using MCM (i.e., Multiple Comparisons with the best) plots (MCMINN, 2004). We used 0.05 as a significance level, so the elements plotted above the gray band were statistically larger than the others. In addition, the dots in the plot represent the median F-Measure value that the algorithms obtained in the projects: a blue dot indicates that the F-Measure distribution

of an algorithm was statistically better than the other algorithms. In contrast, red dots indicate that the F-Measure distribution was not statistically different. To perform this last step, we relied on the `nemenyi` function available in R toolkit.[3]

6.3 ANALYSIS OF THE RESULTS

This section provides an overview of the results for each RQ. We reported the detailed results in our online appendix (MARTINS *et al.*, 2023e).

6.3.1 Classifying Where Developers Would do Test Refactoring

We ran and analyzed 540 different models to classify the code changes in test classes. The Friedman test showed that the *F-Measure* distributions had no statistically significant differences, i.e., p-value>=0.05. However, we still decided to apply the Nemenyi test to analyze which model showed higher values, even if not statistically significant. Figure 6.1 shows the result of the Nemenyi test, where the circle dots are the median likelihood and the error bars indicate the 95% confidence interval. For the interpretation of the results, 60% of likelihood means that a model appears at the top rank for 60% of the projects. We can observe that some balancing techniques failed to generate models, i.e., their distribution is absent in the plot. In addition, balancing techniques did not always provide benefits, as we can assume from the fact that the distributions of the models without balancing techniques are in the middle of the plot. As a result, the best classifier is SVM with Random Undersampling technique.

We reported all the variables that contributed at least once to the predictions to answer the RQ, while our online appendix (MARTINS *et al.*, 2023e) contains the information gain computed for every project. Figure 6.2 shows violin plots concerning Test Code Metrics and Process Metrics. With respect to the Test Code Metrics, we can observe that for the Lines of Code (LOC), Weight Method Class (WMC), and Response for a Class (RFC), most of the values are condensed around the median of the distribution. At the same time, the data for the Number of Methods (NOM) and Assertion Density (AD) follow the distribution. Looking at the Process Metrics, we notice that most metrics follow the distributions, except for Con and MCon (respectively, the Contributors and the Minor Contributors of a commit), in which the values are condensed around 0.

[3] <https://www.r-project.org/>

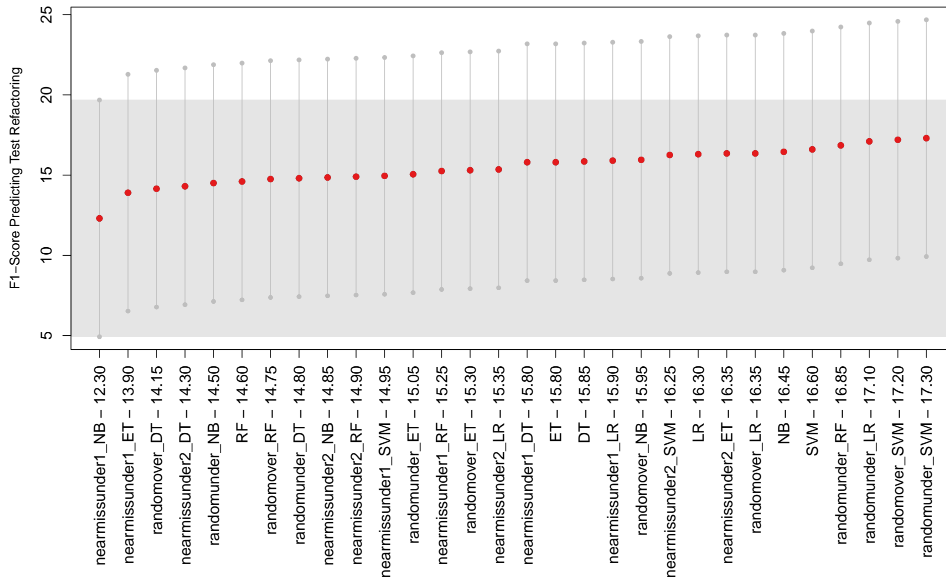


Figure 6.1: The results of the Nemenyi rank applied to the best model with balancing techniques.

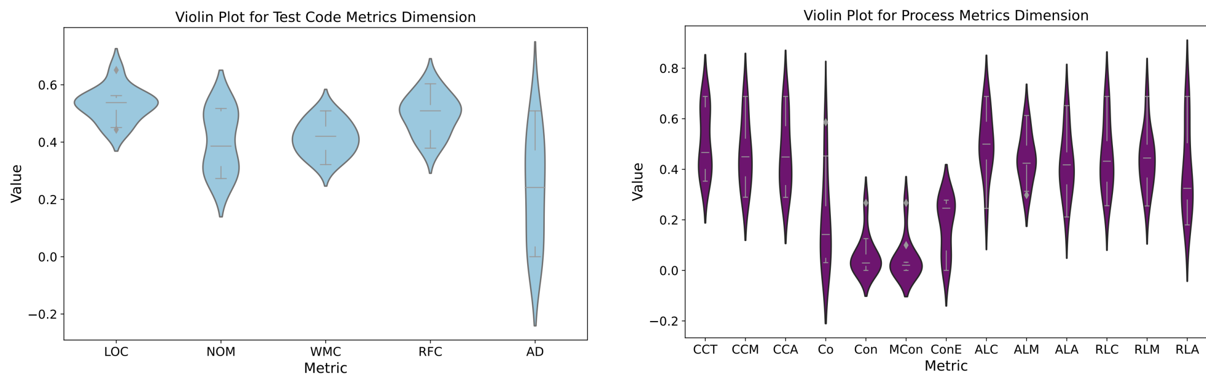


Figure 6.2: Predictive power of Test Code and Process Metrics.

A different discussion can be drawn for Test Smells. Figure 6.3 shows that four test smells (*Default Test*, *Empty Test*, *Verbose Test*, and *Dependent Test*), did not contribute to the model (the predictive power of these metrics was always zero). Then, for most of the test smells in analysis, i.e., *Constructor Initialization* (CI), *General Fixture* (GF), *Ignored Test* (IgT), *Mystery Guest* (MG), *Redundant Assertion* (RA), *Resource Optimism* (RO), and *Sleepy Test* (ST), the values are condensed around zero, so their contribution to the models is minimal. This result suggests that while some smells can be good indicators to classify test code changes, they did not represent the best dimension to use.

Finally, Table 6.2 reports *Precision*, *Recall*, *Accuracy*, and *F-Measure*, for each

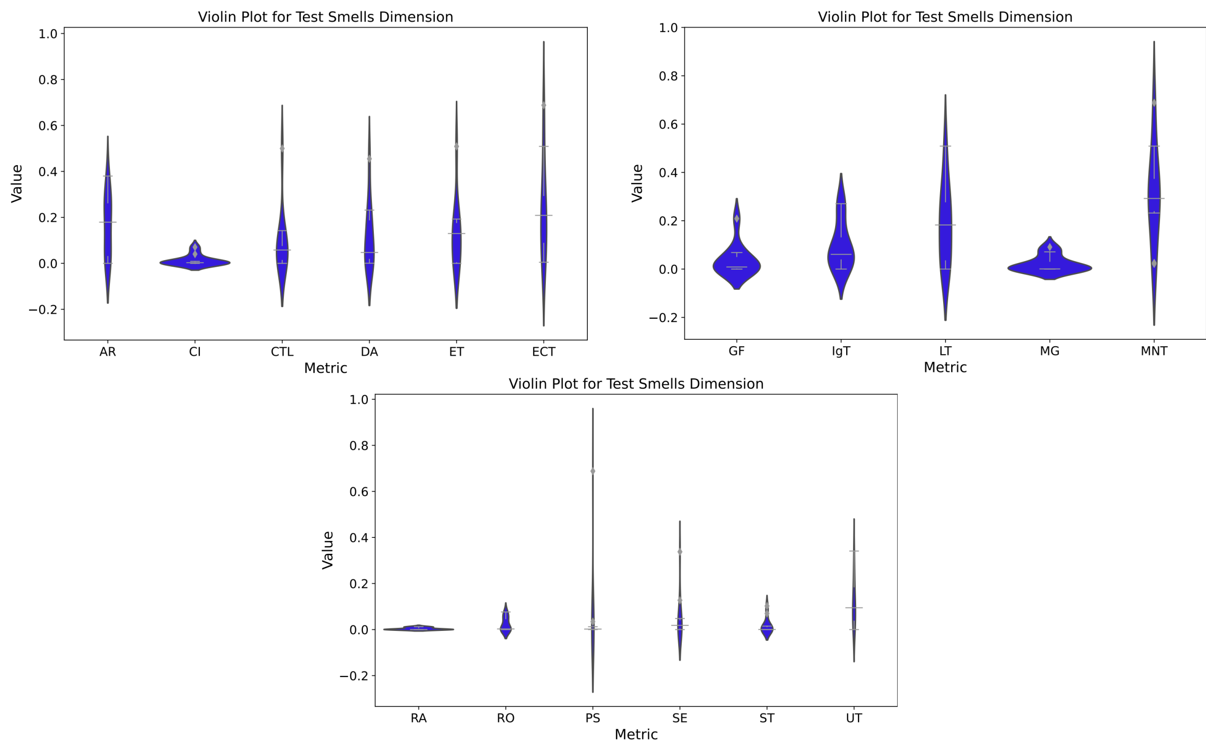


Figure 6.3: Predictive power of Test Smells.

project. The performance obtained varies: in terms of *F-Measure*, the algorithm failed to finish the computation in one case, while for the other projects, the metric ranges between 30% and 100%. For *Precision*, *Recall* and *F-Measure*, the values are generally high: both indicators vary between 20% and 100%. The only exception is represented by HTMLUNIT-DRIVER project: in this case, the indicators are close to zero or NaN. Analyzing such data in detail, we could observe the presence of two projects, i.e., TETRAD and ZANATA-PLATFORM, with a low amount of test classes and test refactoring operations but for which the model could still be built. To further understand the differences between these cases, we analyzed the values for each feature. While we did not find particular differences in test code and process metrics, test smells have a higher information gain in the HTMLUNIT-DRIVER project than in the other two projects under analysis. This result again confirms that test smells are not good indicators for test refactoring operations.

Table 6.2: Performance of the best classifiers for each project analyzed.

Project	TP	FP	TN	FN	Precision	Recall	Acc.	F1
emissary	128	73	13	19	0.63	0.87	0.60	0.73
google-http-java-client	2	6	2	1	0.25	0.66	0.36	0.36
graphhopper	67	63	16	18	0.51	0.78	0.50	0.62
htmlunit-driver	0	2	1	2	0.0	0.0	0.2	NaN
iotdb	4	6	4	3	0.40	0.57	0.47	0.47
itext7	130	35	45	98	0.78	0.57	0.57	0.66
kubernetes-client	8	14	21	23	0.36	0.25	0.44	0.30
questdb	177	193	101	58	0.48	0.75	0.52	0.58
tetrad	2	0	4	0	1.0	1.0	1.0	1.0
zanata-platform	1	2	0	1	0.33	0.50	0.25	0.40

> **Summary$_{3.1}$.** *The best model to predict test code changes is SVM with RANDOM UNDERSAMPLING as balancing technique. The F-Measure obtained varies between 30% and 100%, while the other metrics range between 20% and 100%. The info gain analysis shows that test code and process metrics have higher contributions to the model. Finally, most of the test smells show low contributions (near to zero), and four smells never contributed to the predictions.*

6.3.2 Classifying Specific Test Refactoring Operations

In the context of this research question, we analyzed how supervised ML models classify specific test refactoring operations. Before proceeding, it is important to highlight a consideration that reduced the number of refactorings analyzed. First, we ran 540 models for each test refactoring, for a total of 27,000 models. Once we had collected the results, we observed that for 32 refactorings, not enough information was found to train our models due to the low number of true instances. On the remaining 18 refactorings, for 11 we were able to gather information from only one project, for 4 cases from two projects, and for the last three refactorings, i.e., Modify Method Annotation, Remove Method Annotation, and Add Method Annotation, we gathered information from four, six, and seven projects, respectively.

The Friedman test computed did not show statistically significant differences between the various distributions, and the Nemenyi Test confirmed this result—we reported the plots in our online appendix (MARTINS *et al.*, 2023e). From the statistical tests performed for refactoring operations classified in at least two projects, we observed that the over-sampling techniques, specifically BORDERLINE-SMOTE in the context of Extract

Method detection, worked better than the under-sampling techniques. In addition, for the detection of the Inline Method, Modify Method Annotation, and Move Method, algorithms without any balancing techniques performed better.

Regarding the predictive power of the independent variables, we did not report the violin plot for each refactoring analyzed—the detailed results are in the online appendix (MARTINS *et al.*, 2023e). Regarding the test code metrics dimension, we observed a similar trend to that previously reported, but with slightly lower values: the median of the distributions changed from the range [0.3, 0.5] obtained in $\mathbf{RQ}_{3.1}$, to the range [0.2, 0.4]. Looking at the process metrics dimension, we noticed a different trend in the context of Inline Method classification with respect to $\mathbf{RQ}_{3.1}$. In this case, all the metrics pertaining to this dimension followed the distribution, including the Con and MCon (Contributors and Minor Contributors) metrics. This result may depend on the different number of projects analyzed; in fact, for this test refactoring operation, we collected information only from two projects. Finally, analyzing the Test Smell dimension, we again found that the contribution made by these metrics was minimal.

Table 6.3: Performance obtained to classify specific refactoring operations in QUESTDB.

Precision	Recall	Acc.	F1	Precision	Recall	Acc.	F1	Precision	Recall	Acc.	F1
Extract and Move Method DT - Random Over				Extract Attribute ExT				Extract SUperclass LR - Borderline-SMOTE			
0.20	0.13	0.88	0.16	1.00	0.20	0.97	0.33	0.11	0.14	0.95	0.12
Extract Variable DT				Move Class SVM - Random Over				Pull upo Attribute LR			
0.18	0.54	0.66	0.28	0.10	0.43	0.89	0.15	0.20	0.14	0.93	0.17
Pull up Method ExT				Remove Class Annotation LR - Random Over				Replace Anonymous with Lambda DT - Random Over			
0.08	0.16	0.89	0.11	0.02	0.50	0.88	0.03	0.04	0.14	0.90	0.07
Replace Attribute with Variable SVM Borderline-SMOTE								Split Conditional SVM - Borderline-SMOTE			
0.14	0.50	0.95	0.22					0.25	0.33	0.97	0.28

Tables 6.3 and 6.4 report the performance of the best ML algorithm for the test refactoring operations in analysis. Regarding *F-Measure*, the performance reported in Table 6.3 shows a range between 3% and 33%. Looking at Table 6.4, we can observe that the computation failed in six projects and for four test refactoring operations, while for projects in which the algorithm finished the computation, the *F-Measure* ranges between 19% and 60%. At the same time, we obtained one 100% during the detection of Add

Table 6.4: Performance obtained when classifying the remaining seven refactoring operations.

Project	Precision	Recall	Accuracy	F1
Add Method Annotation — SVM NearMiss2				
google-http-java-client	0.14	0.25	0.25	0.18
htmlunit-driver	0.0	0.0	0.0	NaN
itext7	0.37	0.76	0.60	0.50
questdb	0.07	0.54	0.60	0.13
tetrad	1.0	1.0	1.0	1.0
zanata-platform	0.0	0.0	0.25	0.0
Inline Method — SVM				
questdb	0.10	0.40	0.89	0.16
zanata-platform	0.0	0.0	0.66	0.0
Extract Method — Logistic Regression Borderline-Smote				
itext7	0.06	0.33	0.78	0.10
questdb	0.20	0.66	0.66	0.31
Inline Variable — SVM NearMiss3				
itext7	0.08	0.40	0.59	0.13
questdb	0.06	0.83	0.56	0.11
Modify Method Annotation — SVM				
graphhoopper	0.1	0.33	0.74	0.16
kubernetes-client	0.19	0.37	0.62	0.25
itext7	0.20	0.20	0.76	0.20
zanata-platform	0.0	0.0	0.0	NaN
Move Method — Decision Tree				
htmlunit-driver	0.0	0.0	0.40	NaN
itext7	0.07	0.5	0.81	0.12
Remove Method Annotation — Extra Tree NearMiss1				
emissary	0.49	0.78	0.59	0.60
htmlunit-driver	0.0	0.0	0.33	NaN
iotdb	0.0	0.0	0.64	NaN
itext7	0.16	0.93	0.36	0.28
questdb	0.08	0.89	0.17	0.13
tetrad	0.0	0.0	0.66	NaN
zanata-platform	0.0	0.0	0.0	0.0

Method Annotation in the Tetrad project. A similar discussion can be drawn for the *Precision*: we can observe several 0% and values that did not exceed 50%, except for two projects in two different cases, in which the *Precision* reached 100% (Questdb in Extract Attribute classification and Tetrad in Add Method Annotation classification). Finally, looking at *Recall* and *Accuracy*, the values range between 13% and 97%. These

results were somewhat expected because the number of specific refactorings to classify is low with respect to $\mathbf{RQ}_{3.1}$, in which we have classified the test code changes. Therefore, ML algorithms did not have enough information to train the models. This is the case of Htmlunit-driver project, which has only one true instance of Add Method, Move Method, and Remove Method Annotation refactorings, or the Tetrad and iotdb projects with only one true instance for Remove Method Annotation refactoring.

> **Summary$_{3.2}$.** *When we classify specific test refactoring operations, performance decreases due to a lower amount of data. Both under- and over-sampling techniques contribute to ML models except for seven refactorings, i.e., Extract Attribute, Extract Variable, Pull Up Attribute, Pull Up Method, Inline Method, Modify Method Annotation, and Move Method. Finally, regarding info gain analysis, the results are similar to those already discussed in* $\boldsymbol{RQ}_{3.1}$.

6.4 DISCUSSION

In the following, we first discuss our findings in relation to the work proposed by Aniche et al. (ANICHE *et al.*, 2022). Moreover, we report some points worthy of further analysis and discussion, which we elaborate on in this section.

6.4.1 Relation with Previous Work

Our work was inspired by the one proposed by Aniche *et al.* (2022) in the context of code refactoring. However, there are differences in terms of the dataset employed, the approach taken, and the results achieved that are worth discussing below.

Starting from the data, while Aniche *et al.* (2022) proposed a large-scale empirical study with a dataset comprising over two million refactorings from 11,149 real-world projects from different ecosystems, our work is a preliminary study on a smaller sample in terms of refactorings (826) and projects (10). Nonetheless, our work represents the first attempt to analyze the extent to which supervised ML can be employed in the context of test code refactoring recommendations: as such, it explicitly targets the peculiarities of test code, investigating the effect of specific test code predictors on refactoring types that developers may apply on test code.

Several differences exist in the research method employed. First, we performed a

walk-forward validation (FALESSI *et al.*, 2020) by preserving the chronological order of the data, while Aniche *et al.* (2022) used a stratified 10-fold cross-validation. A second difference is related to the data balancing techniques analyzed. We performed a more comprehensive analysis of the data balancing step by experimenting and statistically verifying several under- and over-sampling techniques. While in the context of $\mathbf{RQ}_{3.1}$ we found that Random Undersampling is used in the best classifier, the result changed in the context of $\mathbf{RQ}_{3.2}$, where other balancing techniques have shown themselves to be more relevant, i.e., NearMiss and Borderline-SMOTE algorithms. The use of multiple balancing techniques combined with the use of statistical tests to evaluate the performance obtained may have contributed to the different results in terms of the best ML model compared with the work of Aniche *et al.* (2022).

Finally, looking at the results, we obtained performance lower with respect to Aniche *et al.* (2022). This represents something expected because of the different number of instances analyzed and the lower number of projects. In addition, we observed that for four specific refactoring operations, i.e., Inline Method, Inline Variable, Extract Method, and Move Method, the performance is lower than those obtained by Aniche *et al.* (2022), but promising. We believe that our study poses the basis for further investigation on the matter.

6.4.2 On the Features and their Predictive Power

Our study analyzed test codes, test smells, and process metrics as features. We chose the features based on the previous research on test code quality (SPADINI *et al.*, 2018; CATOLINO *et al.*, 2019; PECORELLI; PALOMBA; LUCIA, 2021; KIM; CHEN; YANG, 2021). When deciding on those features, we conjectured the structure of a class, the presence of smells, and the developer's experience are crucial factors developers consider when identifying test code to refactor. The performance obtained in the context of $\mathbf{RQ}_{3.1}$ confirms that conjecture. Despite this, it is important to emphasize that the test smell dimension metrics have the lowest or even zero values. This observation implies that although previous studies suggest refactoring when a smell is detected, this is not a factor that leads the developer to refactor.

Interestingly, features that make more sense to humans (e.g., the number of test class lines or methods) contribute most to model building. Moreover, while test code metrics can capture the structure of an element, process metrics can capture its evolution

history. The selection of the metrics suggests, once again, that refactoring represents a human activity. While previous works (KIM; ZIMMERMANN; NAGAPPAN, 2012; SILVA; TSANTALIS; VALENTE, 2016; PALOMBA *et al.*, 2018) investigated those aspects in production refactoring, we noticed a lack of knowledge of the motivations leading developers to refactor test code. Our work poses the basis for further investigating the socio-technical factors influencing the developer's activities in test code refactoring.

6.4.3 ML Models for Test Refactoring Recommendations: How Far Can We Go?

While the performance decrease observed in $\mathbf{RQ}_{3.2}$ was somehow expected because of the fewer instances available for specific refactoring operations, we also noticed that such a decrease reflects a more fundamental problem connected to supervised ML algorithms. To better explain this point, let us discuss the Add Method Annotation refactoring recommendation case in the HTMLUNIT-DRIVER and ITEXT7 projects.

Looking at Table 6.3, we can observe contrasting performance indicators (NaN versus 45% regarding the *F-Measure* metric) when considering the RF algorithm. It is not due to the number of instances of the Add Method Annotation refactoring (which represent $\sim$ 25% of the entire dataset in both cases) but to the total number of test classes used to train the model (8 versus 332). This indicates the lack of data to properly feed recommendation systems based on supervised ML. This observation has three key implications. First, our work suggests that supervised ML algorithms present a good performance in specific cases, namely when there is a sufficient amount of test classes (at least 10 test classes based on our study): in this sense, novel screening mechanisms should be able to analyze the context of a project to establish whether ML solutions would be worth might be devised. Second, our results indicate that future research efforts might be devoted to defining mechanisms through which synthetic data should be created to enable an effective training of ML models. Last but not least, our work stimulates further studies on the capabilities of supervised ML for test refactoring recommendations and, more importantly, on how to complement those recommenders with alternative methods, e.g., heuristic approaches.

6.5 THREATS TO VALIDITY

This section discusses the potential threats that can affect the validity of our empirical study.

Construct validity. The main threat related to the relationship between theory and observation concerns possible imprecision in the data used as the dependent variable in the study. We relied on the TestRefactoringMiner tool (MARTINS *et al.*, 2023a), which is already validated in the literature, making us confident of the reliability of the data. As for the set of independent variables, we chose a mix of metrics capturing the size and structure of test codes and process characteristics. For instance, we know the possible introduction of noise regarding false positive codes and test smells. To partially mitigate this threat, we collected metrics through previously evaluated and well-established tools, showing good accuracy (SPADINI; ANICHE; BACCHELLI, 2018; PECORELLI *et al.*, 2020b; PERUMA *et al.*, 2020a).

Internal Validity. The main threat that could affect the variables and relationships under investigation concerns the imbalanced dataset, i.e., there are more instances of the non-refactoring class in our dataset than instances of the refactoring class. Therefore, we analyzed several under- and over-sampling techniques to understand their impact on the data and performance.

External Validity. The main threat concerning the generalization of results is the selection of subject projects. We selected ten open-source Java projects from GitHub, only a fraction of the complete picture of open-source Java projects. Consequently, we can not generalize the results to distinct domains, industrial projects, and other programming languages. Therefore, replications of this study would corroborate our findings in different contexts. Our appendix (MARTINS *et al.*, 2023e) provides all materials and scripts used in this study to stimulate further research.

Conclusion validity. Threats of this category are related to the relationship between treatment and outcome. A key potential source of bias can have been related to the presence of independent variables providing a similar contribution to the performance of the experimented models (O'BRIEN, 2007). To account for this threat, we computed the information gain provided by each feature used to feed the models. That computation allowed us to verify the independent variables were orthogonal, contributing to the models built. To further corroborate the conclusions drawn in the study, we applied the

Friedman and Nemenyi tests (NEMENYI, 1963; SHELDON; FILLYAW; THOMPSON, 1996), which allowed us to report our findings from a statistical perspective.

Our work represents the first attempt to recommend test refactoring operations. As such, we experimented with multiple techniques to identify the best algorithm. Our online appendix (MARTINS *et al.*, 2023e) includes all our findings, which researchers can use to understand further the impact of ML techniques.

6.6 CHAPTER SUMMARY

Our results show that using ML to classify test code changes is feasible. Still, the performance decreases when classifying specific test refactoring operations due to the lower amount of test classes analyzed. Our qualitative analysis shows the need to explore the socio-technical factors influencing the developer's activities. To sum up, our main contributions are: (1) we devised a ML approach to classify the developers' intention to apply test refactoring and the test-specific operations applied, and (2) we released a publicly available replication package (MARTINS *et al.*, 2023e) with data, scripts, and results of our experiment.

In this chapter, we performed an empirical study to assess the performance of supervised ML algorithms to classify test code refactoring operations in 10 JAVA projects. In $\mathbf{RQ}_{3.1}$, we evaluated the performance of six ML algorithms to classify specific test code changes in which practitioners would do test refactoring using test smells, test metrics, and process variables, i.e., code churn, as predictors. As a result, we found that the Support Vector Machine (SVM) algorithm outperforms the others, with performance varying between 30% to 100%. In $\mathbf{RQ}_{3.2}$, we evaluated the classification of specific test refactoring operations, and we found that performance decreases due to a lower amount of data. The discussion focused on the predictive power of the features as well as the feasibility of the approach, suggesting the need to change our current vision of test refactoring.

As for our future work, we aim to continue exploring the capabilities of supervised ML in providing recommendations for test refactoring. In addition, we can investigate the generation of synthetic data for training ML models and the influence of socio-technical factors on developers' activities in test code refactoring.

Chapter 7

CONCLUSIONS

This thesis investigated how developers refactor the test code to fix test smells in practice and what factors drive developers to refactor the test code. In addition, we explored Machine Learning (ML) techniques to classify whether developers would apply refactorings in the test code and which test-specific refactoring operations they would apply. By considering the developers' perspective, we aim to gain ground to elaborate on more interactive refactoring techniques that can help overcome current limitations related to the lack of support for test-specific refactorings and the number of false positives detected by automated refactoring recommendation tools.

This chapter presents the final remarks of this thesis. Section 7.1 summarizes the results of our research. Section 7.2 lists the main scientific contributions we have achieved so far. Section 7.3 elaborates our vision for future work.

7.1 RESULTS ADDRESSING OUR GOAL AND RESEARCH QUESTIONS

To answer our goal, we present the summary of results for our high-level research questions (RQs) in the following.

RQ_1. How do developers perform test code refactorings to fix test smells in open-source projects? To answer our **RQ_1**, we started by manually classifying changes performed by developers in the test code of 13 open-source Java projects of the Apache Foundation. We classified the changes into test smells and refactorings. Then,

we compiled a catalog of test-specific refactorings to fix test smells, compared the catalog with state-of-the-art catalogs, and gathered developers' feedback on the usefulness of our test-specific refactorings. In summary, the main outcomes include:

- We identified 9 test smells refactored in the projects. Although we can widely find the investigation of eight test smells in the literature (the ECT, AR, BaN, IgT, RP, EpT, UT, and ST test smells), we found one test smell still not investigated (Inappropriate Assertion - InA);
- We identified 11 test refactorings in practice and literature to fix test smells. Although we can widely find the use of three refactorings in practice (*Replace try/-catch with @Test annotation*, *Split assertions into single methods*, and *Surround assertions with assertThrows*), we found another three refactorings not previously defined in the literature (*Replace NOT operator*, *Split Conditional into statements*, and *Replace Reserved Words*);
- When asking for developers' feedback on the set of test-specific refactorings, most showed appreciation for the efforts to improve test quality. They also considered a historical context to highlight that certain constructs of the testing framework were unavailable when creating the tests. Still, there are better ways to implement the same test. In addition, some developers showed awareness of current best practices and proactiveness to improve the test code.

$\mathbf{RQ_2}$. How do test refactoring operations affect test code quality? In our **$\mathbf{RQ_2}$**, we collected a dataset containing test smells, test refactorings, and structural metrics from 63 open-source JAVA projects. Next, we investigated whether developers targeted low-quality test code when refactoring and whether the refactorings improved the code quality. In summary, we could observe that:

- Structural metrics and test smells influence the likelihood of developers applying most test refactorings. In particular, the AsD metric and the DA test smell hold significance for most test refactorings than the other metrics and test smells;
- Test refactorings derived from Fowler's catalog (FOWLER, 1999) and some test-specific refactorings (*Replace Rule w/ assertThrows*, *Replace Conditional by ParameterizedTest*, and *Replace @Test with assertThrows*) were responsible for most improvements in the test code regarding its complexity, size, and coupling. The

same test-specific refactorings also influenced fixing the GF, LT, AR, and ECT test smells.

RQ_3: How accurately can we suggest test refactoring operations for fixing test smells using ML techniques? To answer **RQ_3**, we collected test smells, test refactorings, and process and structural metrics from 10 open-source JAVA projects. Subsequently, we modeled a binary classification problem to investigate whether developers would refactor the test class in the projects using ML algorithms (Random Forest - RF, Decision Trees - DT, Extra-Tree - ExT, Logistic Regression - LR, Naive Bayes - NB, and Support Vector Machine - SVM). Similarly, we modeled a binary classification problem to classify whether developers would apply a specific test refactoring operation. Our results show that:

- The SVM algorithm was the best ML algorithm, and the AR, CTL, DA, ET, ECT, LT, MNT, PS, SE, and UT test smells can be good predictors to classify whether developers would perform test refactorings. Yet, structural and process metrics contributed more to the models than test smells;
- The best ML algorithm varied for analyzing specific refactoring operations. In particular, the RF algorithm presented the best results for the *Add Method Annotation* refactoring, the DT algorithm presented the best results for the *Inline Variables* refactoring, the ExT algorithm presented the best results for the *Remove Method Annotation* refactoring, the NB algorithm presented the best results for the *Extract Method* and *Move Method* refactorings, and the SVM algorithm resented the best results for the *Inline Method* and *Modify Method Annotation* refactorings;
- The classification of whether developers would refactor the test code varied in different projects (from 0% to 100% of precision and recall scores, with average values of 47% and 60%, respectively). However, the performance decreased in classifying specific test refactoring operations due to less data. It indicates supervised ML algorithms worked when a sufficient amount of test classes exists, calling for more investigations regarding searching for ML solutions specific to the specific context of projects, creating synthetic data to enable the training of ML models, and complementing models with alternative approaches.

Now, back to the research objective of **investigating the feasibility of identifying test refactoring opportunities and proper fixings based on ML techniques**,

our approach demonstrated promise to classify specific test refactorings based on structural metrics and test smells.

7.2 CONTRIBUTIONS

This section presents our main contributions regarding research investigations, supplementary materials and tools, and academic publications.

7.2.1 Research Contribution

The main contributions of this work are:

1. **Body of knowledge on test code refactorings and test smells.** The number of studies on test smells is increasing year by year. We performed ad-hoc reviews to synthesize state-of-the-art evidence on test code refactorings and test smells for use in various investigations in the field;
2. **Catalog with common test smells and test refactorings in practice.** We manually analyzed modified test files of open-source projects to compile a catalog of test smells and the refactoring operations used to fix them based on actual development practices;
3. **Identifying refactoring opportunities in test code and proper fixings for test smells.** We developed ML models to learn from the features extracted from the change history of the test code, including process metrics, structural metrics, and test smells. These models were designed to classify the code changes where developers would undertake test refactoring and to predict which specific test refactoring operations developers would apply.

7.2.2 Materials and Tools

We also contributed to the community to support the investigations with computational tools, datasets, and other materials. The main contributions are:

- **Extension of RefactoringMiner.** We built theTestRefactoringMiner tool on top of the RefactoringMiner tool (TSANTALIS; KETKAR; DIG, 2020).

We extended the set of changes reported by the tool to detect the refactorings: 1) *Add explanation message*, 2) *Replace reserved words*, 3) *Split conditional parameters*, 4) *Replace the not (!) operator*, 5) *Replace try/catch with asserThrows*, 6) *Replace @Rule annotation with assertThrows*, and 7) *Replace @Test annotation with assertThrows*. The tool is made available in a GitHub repository[1];

- **Extension of VITRuM tool.** The VITRuM tool is a plugin for IntelliJ IDE and calculates structural metrics (e.g., the AsD, LOC, NOM, RFC, and WMC metrics) and coverage and mutation scores. We changed it to allow it to collect data on a large scale, as its original version is IDE-dependent. The tool adaptation is available in a GitHub repository[2];

- **Extension of tsDetect tool.** The tsDetect tool works via the command line and requires the execution of three different modules to generate the final results containing the test smells per class. We changed it to execute all modules without manual intervention and added functionality to clone projects automatically, allowing the tool to collect data on a large scale. The tool adaptation is available in a GitHub repository[3];

- **Datasets.** First, we released a curated dataset containing 611 instances of pairs of test code, with test smells and their respective test refactorings (MARTINS *et al.*, 2023b). Subsequently, we extended the TestRefactoringMiner, tsDetect, and VITRuM tools to collect large-scale datasets combining information on test smells, test refactorings, and structural metrics for 63 projects (MARTINS *et al.*, 2023d). Last, we created a smaller dataset to add information on process metrics for ten projects (MARTINS *et al.*, 2023e);

- **Catalog of test-specific refactorings.** We proposed the TSR-Catalog (Catalog of Test Smells Refactorings). That catalog outlines the reengineering process for conducting test-specific refactorings to remove test smells. Initially, it possesses information on test smells and test refactorings manually classified from 375 test files from the Apache Foundation. The catalog is available online[4].

[1] Available at: https://github.com/arieslab/TestRefactoringMiner
[2] Available at: https://github.com/luana-martins/MyVITRuM
[3] Available at: https://github.com/luana-martins/tsDetectExtended
[4] Available at: https://tsr-catalog.readthedocs.io/en/latest/

7.2.3 Academic contributions

Next, we list the set of papers resulting from this investigation:

- Luana Almeida Martins, Taher Ahmed Ghaleb, Heitor A. X. Costa, Ivan Machado: A comprehensive catalog of refactoring strategies to handle test smells in Java-based systems. Softw. Qual. J. 32(2): 641-679 (2024)
- Luana Almeida Martins, Heitor A. X. Costa, Ivan Machado: On the diffusion of test smells and their relationship with test code quality of Java projects. J. Softw. Evol. Process. 36(4) (2024)
- Luana Almeida Martins, Heitor A. X. Costa, Márcio Ribeiro, Fabio Palomba, Ivan Machado: Automating Test-Specific Refactoring Mining: A Mixed-Method Investigation. SCAM 2023: 13-24
- Luana Martins, Valeria Pontillo, Heitor Costa, Filomena Ferrucci, Fabio Palomba, Ivan Machado. 2023. Test Code Refactoring Unveiled: Where and How Does It Affect Test Code Quality and Effectiveness? In Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution, Registered Reports track, 1-9
- Luana Almeida Martins, Denivan Campos, Railana Santana, Joselito Mota Júnior, Heitor A. X. Costa, Ivan Machado: Hearing the voice of experts: Unveiling Stack Exchange communities' knowledge of test smells. CHASE 2023: 80-91
- Luana Almeida Martins, Carla I. M. Bezerra, Heitor A. X. Costa, Ivan Machado: Smart prediction for refactorings in the software test code. SBES 2021: 115-120

Other important publications from this thesis, in which there are other main (first) authors but still hold the importance for this work:

- Railana Santana, Luana Almeida Martins, Tássio Virgínio, Larissa Rocha, Heitor A. X. Costa, Ivan Machado: An empirical evaluation of RAIDE: A semi-automated approach for test smells detection and refactoring. Sci. Comput. Program. 231: 103013 (2024)

- Tássio Virgínio, Luana Almeida Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor A. X. Costa, Ivan Machado: On the test smells detection: an empirical study on the JNose Test accuracy. J. Softw. Eng. Res. Dev. 9: 8:1-8:14 (2021)

- Nildo Silva Junior, Luana Almeida Martins, Larissa Rocha, Heitor A. X. Costa, Ivan Machado: How are test smells treated in the wild? A tale of two empirical studies. J. Softw. Eng. Res. Dev. 9: 9:1-9:16 (2021)

- Denivan Campos, Luana Almeida Martins, Carla I. M. Bezerra, Ivan Machado: Investigating Developers' Contributions to Test Smell Survivability: A Study of Open-Source Projects. SAST 2023: 86-95

- Denivan Campos, Luana Almeida Martins, Ivan Machado: An empirical study on the influence of developers' experience on software test code quality. SBQS 2022: 3:1-3:10

- Railana Santana, Luana Almeida Martins, Tássio Virgínio, Larissa Rocha Soares, Heitor A. X. Costa, Ivan Machado: Refactoring Assertion Roulette and Duplicate Assert test smells: a controlled experiment. CIBSE 2021: 1-16.

- Nildo Silva Junior, Larissa Rocha Soares, Luana Almeida Martins, Ivan Machado: CIBSE 2020: 462-475

- Tássio Virgínio, Luana Almeida Martins, Larissa Rocha Soares, Railana Santana, Heitor A. X. Costa, Ivan Machado: An empirical study of automatically-generated tests from the perspective of test smells. SBES 2020: 92-96

- Railana Santana, Luana Almeida Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor A. X. Costa, Ivan Machado: RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring. SBES 2020: 374-379

- Tássio Virgínio, Luana Almeida Martins, Larissa Rocha Soares, Railana Santana, Adriana Cruz, Heitor A. X. Costa, Ivan Machado: JNose: Java Test Smell Detector. SBES 2020: 564-569

7.3 FUTURE RESEARCH DIRECTIONS

This section points out avenues for future research regarding automating test refactoring.

Empirical validation with developers. The proposed catalog of test smells and refactoring strategies can further benefit from empirical validation with software practitioners in the field, including surveys, interviews, or focus groups of practitioners to gather their feedback and experiences with the identified refactorings. In addition, in-depth case studies with developers can offer insights into the challenges developers could face, the reasoning behind refactoring decisions, and observed benefits after refactoring.

Experimentation with real-world projects. While our research identified test smells and their respective refactorings in real-world projects, our findings can be context-specific to the projects we selected. To enhance the external validity of our research, conducting more extensive experiments or case studies involving industry practitioners would be beneficial, helping to generalize the results.

Assessment of refactoring impact on quality attributes. Future research is encouraged to perform controlled experiments to assess the impact of the proposed refactorings on key quality attributes, such as test reliability, maintainability, and execution time. Such experiments are crucial for providing evidence of the practical advantages of test smell refactorings compared to alternative approaches or the option of not refactoring.

Investigating the generation of synthetic data for model training. While our approach holds promise in identifying refactoring opportunities and proper fixings, it is limited due to the number of refactorings composing the datasets. The investigation of methods for creating realistic and diverse synthetic data can help address limitations in the data availability, improving the generalization and robustness of the models.

Analyzing socio-technical factors in developers' test code refactoring activities. The process metrics selected to analyze the developers' activities in the test code contributed the most to the ML models. Therefore, understanding the human and social aspects of test code refactoring can provide valuable insights into improving the adoption and effectiveness of automated recommendations.

Exploring supervised ML algorithms for test refactoring recommendations. We can delve deeper into our investigation to improve the capabilities of ML models. In particular, we could refine existing models, explore different algorithms, explore different contexts of software projects, and incorporate additional features to enhance the accuracy and relevance of the recommendations.

REFERENCES

AFONSO, J.; CAMPOS, J. Automatic generation of smell-free unit tests. In: IEEE. *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. [S.l.], 2023. p. 9–16.

ALJEDAANI, W.; PERUMA, A.; ALJOHANI, A.; ALOTAIBI, M.; MKAOUER, M. W.; OUNI, A.; NEWMAN, C. D.; GHALLAB, A.; LUDI, S. Test smell detection tools: A systematic mapping study. In: *Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (EASE 2021), p. 170–180.

AMANNEJAD, Y.; GAROUSI, V.; IRVING, R.; SAHAF, Z. A search-based approach for cost-effective software test automation decision support and an industrial case study. In: IEEE. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. New York, NY, USA, 2014. p. 302–311.

ANICHE, M.; MAZIERO, E.; DURELLI, R.; DURELLI, V. H. S. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, v. 48, n. 4, p. 1432–1450, 2022.

APACHE ACCUMULO. *Remove unused code*. 2019. Available at: <https://github.com/apache/accumulo/commit/f10b4073dba6b8095e9934e9ea158eb9f45c6f67>. Last access: 11-08-2023.

APACHE ACCUMULO. *Backport JUnit upgrade from #1562 to 1.10.1*. 2021. Available at: <https://github.com/apache/accumulo/commit/d4fd27f32dc2611a23f67b1d3e8dafd8ee05a1cb>. Last access: 11-08-2023.

APACHE CAMEL. *CAMEL-11807: Migrated camel-telegram tests to JUnit 5*. 2019. Available at: <https://github.com/apache/camel/commit/626196af0baf18a859c55bdf91526b447b367faf>. Last access: 11-08-2023.

APACHE CAMEL. *CAMEL-13629: Renamed tests to follow *Test pattern*. 2019. Available at: <https://github.com/apache/camel/commit/9dc4dc6cd2c6cee75892e9a57105d79bfdcc8f5c>. Last access: 11-08-2023.

APACHE CAMEL. *Camel-Pulsar: Removed System.out statements from tests*. 2020. Available at: <https://github.com/apache/camel/commit/f7d1dbbf736e8b50ac5f17e5d25829a0a6aa5d4e>. Last access: 11-08-2023.

APACHE CAMEL. *Fixed missing assertions in test code #6002)*. 2020. Available at: <https://github.com/apache/camel/commit/7a43633b3c3587d949724f580ad0015a6f65ef82>. Last access: 11-08-2023.

APACHE CAMEL. *core/camel-core: replace Thread.sleep with Awaitility in tests*. 2021. Available at: <https://github.com/apache/camel/commit/d58c7318cb81f8faa5f2f4acd28d7a215855450d>. Last access: 11-08-2023.

APACHE CAMEL. *Test-cleanups: fixed incorrect assertions for exceptions*. 2021. Available at: <https://github.com/apache/camel/commit/c30deabcaed4726bce4371d76257db63f2eba87c>. Last access: 11-08-2023.

APACHE CXF. *More test assertion cleanup*. 2019. Available at: <https://github.com/apache/cxf/commit/7e11da7a566a95adc64143c0575b7ef86e0fbe5a>. Last access: 11-08-2023.

APACHE CXF. *Using assertEquals instead of assertTrue in some of the tests*. 2019. Available at: <https://github.com/apache/cxf/commit/4955ca652f16e781524612383af27c650e10cbdc>. Last access: 11-08-2023.

APACHE CXF. *Using assertEquals instead of assertTrue in some of the tests*. 2019. Available at: <https://github.com/apache/cxf/commit/4955ca652f16e781524612383af27c650e10cbdc>. Last access: 11-08-2023.

APACHE KAFKA. *KAFKA-12819: Add assert messages to MirrorMaker tests plus other quality of life improvements*. 2021. Available at: <https://github.com/apache/kafka/commit/56d9482462c2aa941b151015499fc59485fe7426>. Last access: 11-08-2023.

APACHE KAFKA. *MINOR: Optimize assertions in unit tests (#9955)*. 2021. Available at: <https://github.com/apache/kafka/commit/f4c2030b2006fc0c447a10f8b251579424f39f7b>. Last access: 11-08-2023.

ARANEGA, V.; DELPLANQUE, J.; MARTINEZ, M.; BLACK, A. P.; DUCASSE, S.; ETIEN, A.; FUHRMAN, C.; POLITO, G. Rotten green tests in java, pharo and python: An empirical study. *Empirical Software Engineering*, Springer, v. 26, n. 6, p. 1–41, 2021.

AZEEM, M. I.; PALOMBA, F.; SHI, L.; WANG, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, Elsevier, v. 108, p. 115–138, 2019.

BAKER, P.; EVANS, D.; GRABOWSKI, J.; NEUKIRCHEN, H.; ZEISS, B. Trex-the refactoring and metrics tool for ttcn-3 test specifications. In: IEEE. *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*. New York, NY, USA, 2006. p. 90–94.

BAQAIS, A. A. B.; ALSHAYEB, M. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, Springer, v. 28, n. 2, p. 459–502, 2020.

BAVOTA, G.; CARLUCCIO, B. D.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R.; STROLLO, O. When does a refactoring induce bugs? an empirical study. In: IEEE. *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. [S.l.], 2012. p. 104–113.

BAVOTA, G.; De Lucia, A.; Di Penta, M.; OLIVETO, R.; PALOMBA, F. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, v. 107, p. 1–14, 2015.

BAVOTA, G.; LUCIA, A. D.; MARCUS, A.; OLIVETO, R. Recommending refactoring operations in large software systems. *Recommendation Systems in Software Engineering*, Springer, p. 387–419, 2014.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: IEEE. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. New York, NY, USA, 2012. p. 56–65.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A.; BINKLEY, D. Are test smells really harmful? an empirical study. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 20, n. 4, p. 1052–1094, Aug. 2015.

BECK, K. L. *Test-driven Development: by example*. Upper Saddle River, NJ: Addison-Wesley, 2003. (The Addison-Wesley signature series). ISBN 978-0-321-14653-3.

BELL, J.; KAISER, G.; MELSKI, E.; DATTATREYA, M. Efficient dependency detection for safe java test acceleration. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 770–781.

BELL, J.; LEGUNSEN, O.; HILTON, M.; ELOUSSI, L.; YUNG, T.; MARINOV, D. Deflaker: Automatically detecting flaky tests. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. p. 433–444.

BIAGIOLA, M.; STOCCO, A.; MESBAH, A.; RICCA, F.; TONELLA, P. Web test dependency detection. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 154–164.

BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

BLAND, J. M.; ALTMAN, D. G. The odds ratio. *Bmj*, British Medical Journal Publishing Group, v. 320, n. 7247, p. 1468, 2000.

BLESER, J. D.; NUCCI, D. D.; ROOVER, C. D. Assessing diffusion and perception of test smells in scala projects. In: IEEE. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. New York, NY, USA, 2019. p. 457–467.

BLESER, J. D.; NUCCI, D. D.; ROOVER, C. D. Socrates: Scala radar for test smells. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. New York, NY, USA: Association for Computing Machinery, 2019. (Scala '19), p. 22–26.

BODEA, A. Pytest-smell: A smell detection tool for python unit tests. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2022. (ISSTA 2022), p. 793–796.

BOIS, B. D.; DEMEYER, S.; VERELST, J. Refactoring - improving coupling and cohesion of existing code. In: *11th Working Conf. on Reverse Engineering*. [S.l.: s.n.], 2004. p. 144–151.

BOWES, D.; HALL, T.; PETRIC, J.; SHIPPEY, T.; TURHAN, B. How good are my tests? In: *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*. New York, NY, USA: IEEE Press, 2017. (WETSoM '17), p. 9–14.

BREIMAN, L. Random forests. *Mach. Learn.*, Kluwer Academic Publishers, USA, v. 45, n. 1, p. 5–32, Oct. 2001.

BREUGELMANS, M.; ROMPAEY, B. V. Testq: Exploring structural and maintenance characteristics of unit test suites. In: IEEE. *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. New York, NY, USA, 2008.

CAMPOS, D.; MARTINS, L.; BEZERRA, C.; MACHADO, I. Investigating developers' contributions to test smell survivability: A study of open-source projects. In: *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*. New York, NY, USA: Association for Computing Machinery, 2023. p. 86–95.

CAMPOS, D.; MARTINS, L.; MACHADO, I. An empirical study on the influence of developers' experience on software test code quality. In: *Proceedings of the XXI Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2023.

CAMPOS, D.; ROCHA, L.; MACHADO, I. Developers perception on the severity of test smells: an empirical study. In: *Iberoamerican Conference on Software Engineering*. Costa Rica: arxiv, 2021. p. 1–14.

CATOLINO, G.; PALOMBA, F.; ZAIDMAN, A.; FERRUCCI, F. How the experience of development teams relates to assertion density of test classes. In: *ICSME 2019*. Cleveland,USA: IEEE, 2019. p. 223–234.

CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; MELLO, R. de; FONSECA, B.; RIBEIRO, M.; CHáVEZ, A. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 465–475. ISBN 9781450351058.

CHáVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; GARCIA, A. How does refactoring affect internal quality attributes? a multi-project study. In: *Proceedings of the XXXI Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (SBES '17), p. 74–83. ISBN 9781450353267.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, v. 16, p. 321–357, 2002.

CHEN, Z.; EMBURY, S. M.; VIGO, M. Who is afraid of test smells? assessing technical debt from developer actions. In: SPRINGER. *IFIP International Conference on Testing Software and Systems*. [S.l.], 2023. p. 160–175.

COHEN, J. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, Sage Publications Sage CA: Thousand Oaks, CA, v. 20, n. 1, p. 37–46, 1960.

CORTES, C.; VAPNIK, V. Support-vector networks. *Mach. Learn.*, Kluwer Academic Publishers, USA, v. 20, n. 3, p. 273–297, Sep. 1995.

COUNSELL, S.; HIERONS, R. M. Refactoring test suites versus test behaviour: A ttcn-3 perspective. In: *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*. New York, NY, USA: Association for Computing Machinery, 2007. (SOQUA '07), p. 31–38.

CRUZ, A.; COSTA, H. Uma abordagem visual para evolução de test smells em sistemas de software java. In: *Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática*. Porto Alegre, RS, Brasil: SBC, 2020. p. 63–69.

DALLAL, J. A. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *IST*, Elsevier, v. 58, p. 231–249, 2015.

DAMASCENO, H.; BEZERRA, C.; CAMPOS, D.; MACHADO, I.; COUTINHO, E. Test smell refactoring revisited: What can internal quality attributes and developers' experience tell us? *Journal of Software Engineering Research and Development*, p. 13–1, 2023.

DAMASCENO, H.; BEZERRA, C.; COUTINHO, E.; MACHADO, I. Analyzing test smells refactoring from a developers perspective. In: *Proceedings of the XXI Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '22). ISBN 9781450399999.

DELPLANQUE, J.; DUCASSE, S.; POLITO, G.; BLACK, A. P.; ETIEN, A. Rotten green tests. In: *Proceedings of the 41st International Conference on Software Engineering*. New York, NY, USA: IEEE Press, 2019. (ICSE '19), p. 500–511.

DEURSEN, A.; MOONEN, L. M.; BERGH, A.; KOK, G. *Refactoring Test Code*. NLD, 2001.

DI, Z.; LI, B.; LI, Z.; LIANG, P. A preliminary investigation of self-admitted refactorings in open source software. In: KSI RESEARCH INC. AND KNOWLEDGE SYSTEMS INSTITUTE GRADUATE SCHOOL. *Int.l Conf. on Software Engineering and Knowledge Engineering*. [S.l.], 2018. v. 2018, p. 165–168.

FALESSI, D.; HUANG, J.; NARAYANA, L.; THAI, J. F.; TURHAN, B. On the need of preserving order of data when validating within-project defect classifiers. *Empirical Software Engineering*, Springer, v. 25, p. 4805–4830, 2020.

FATIMA, S.; GHALEB, T. A.; BRIAND, L. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*, IEEE, 2022.

FERNANDES, D.; MACHADO, I.; MACIEL, R. Handling test smells in python: Results from a mixed-method study. In: *Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. p. 84–89.

FERNANDES, D.; MACHADO, I.; MACIEL, R. Tempy: Test smell detector for python. In: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022. (SBES '22), p. 214–219. ISBN 9781450397353.

FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; UCHôA, A.; BIBIANO, A. C.; GARCIA, A.; CORREIA, J. a. L.; SANTOS, F.; NUNES, G.; BARBOSA, C.; FONSECA, B.; MELLO, R. de. The buggy side of code refactoring: Understanding the relationship between refactorings and bugs. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA: Association for Computing Machinery, 2018. (ICSE '18), p. 406–407. ISBN 9781450356633. Available at: <https://doi.org/10.1145/3183440.3195030>.

FOWLER, M. *Refactoring - Improving the Design of Existing Code*. Upper Saddle River, NJ: Addison-Wesley, 1999. (Addison Wesley object technology series).

FRASER, G.; GAMBI, A.; ROJAS, J. M. Teaching software testing with the code defenders testing game: Experiences and improvements. In: IEEE. *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. New York, NY, USA, 2020. p. 461–464.

FREUND, Y.; MASON, L. The alternating decision tree learning algorithm. In: *Proceedings of the Sixteenth Int.l Conf. on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. p. 124–133. ISBN 1558606122.

FULCINI, T.; GARACCIONE, G.; COPPOLA, R.; ARDITO, L.; TORCHIANO, M. Guidelines for gui testing maintenance: a linter for test smell detection. In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. New York, NY, USA: Association for Computing Machinery, 2022. p. 17–24.

FUSHIHARA, Y.; AMAN, H.; AMASAKI, S.; YOKOGAWA, T.; KAWAHARA, M. A trend analysis of test smells in python test code over commit history. In: *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. [S.l.: s.n.], 2023. p. 310–314.

GAMIDO, H. V.; GAMIDO, M. V. Comparative review of the features of automated software testing tools. *International Journal of Electrical and Computer Engineering*, IAES Institute of Advanced Engineering and Science, v. 9, n. 5, p. 4473, 2019.

GAROUSI, V.; AMANNEJAD, Y.; BETIN CAN, A. Software test-code engineering: A systematic mapping. *Information and Software Technology*, v. 58, p. 123–147, 2015.

GAROUSI, V.; KüçüK, B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, v. 138, p. 52–81, 2018.

GAROUSI, V.; KüçüK, B.; FELDERER, M. What we know about smells in software test code. *IEEE Software*, v. 36, n. 3, p. 61–73, 2019.

GATRELL, M.; COUNSELL, S.; HALL, T. Empirical support for two refactoring studies using commercial C# software. In: *Proceedings of the 13th International Conference on Evaluation and Assessment in Software Engineering*. Swindon, GBR: BCS Learning and Development Ltd., 2009. (EASE'09), p. 1–10.

GEURTS, P.; ERNST, D.; WEHENKEL, L. Extremely randomized trees. *Machine learning*, Springer, v. 63, p. 3–42, 2006.

GRANO, G.; IACO, C. D.; PALOMBA, F.; GALL, H. C. Pizza versus pinsa: On the perception and measurability of unit test code quality. In: IEEE. *2020 IEEE Int.l Conf. on Software Maintenance and Evolution (ICSME)*. [S.l.], 2020. p. 336–347.

GRANO, G.; PALOMBA, F.; NUCCI, D. D.; LUCIA, A. D.; GALL, H. C. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, Elsevier, v. 156, p. 312–327, 2019.

GREILER, M.; DEURSEN, A. van; STOREY, M.-A. Automated detection of test fixture strategies and smells. In: IEEE. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. New York, NY, USA, 2013. p. 322–331.

GREILER, M.; ZAIDMAN, A.; DEURSEN, A. V.; STOREY, M.-A. Strategies for avoiding text fixture smells during software evolution. In: IEEE. *2013 10th Working Conference on Mining Software Repositories (MSR)*. New York, NY, USA, 2013. p. 387–396.

GUERRA, E. M.; FERNANDES, C. T. Refactoring test code safely. In: IEEE. *International Conference on Software Engineering Advances (ICSEA 2007)*. New York, NY, USA, 2007. p. 44–44.

GYORI, A.; SHI, A.; HARIRI, F.; MARINOV, D. Reliable testing: Detecting state-polluting tests to prevent test dependency. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2015. (ISSTA 2015), p. 223–233.

HADJ-KACEM, M.; BOUASSIDA, N. A multi-label classification approach for detecting test smells over java projects. *Journal of King Saud University-Computer and Information Sciences*, Elsevier, 2021.

HAN, H.; WANG, W.; MAO, B. Borderline-smote: a new over-sampling method in imbalanced data sets learning. In: *Int.l Conf. on intelligent computing*. Hefei China: Springer, 2005. p. 878–887.

HAN, J.; KAMBER, M.; PEI, J. Data mining concepts and techniques third edition. *The Morgan Kaufmann Series in Data Management Systems*, v. 5, n. 4, p. 83–124, 2011.

HE, H.; BAI, Y.; GARCIA, E. A.; LI, S. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In: *Int.l joint Conf. on neural networks*. Hong Kong: IEEE, 2008. p. 1322–1328.

HESSE-BIBER, S. N. *Mixed methods research: Merging theory with practice*. New York, NY, USA: Guilford Press, 2010.

HO, T. K. Random decision forests. In: *Document analysis and recognition, 1995., proceedings of the third Int.l Conf. on*. Montreal, QC, Canada: IEEE, 1995. v. 1, p. 278–282.

HUO, C.; CLAUSE, J. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 621–631.

IANNONE, E.; CODABUX, Z.; LENARDUZZI, V.; LUCIA, A. D.; PALOMBA, F. Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security. *Empirical Software Engineering*, Springer, v. 28, n. 4, p. 89, 2023.

JORGE, D.; MACHADO, P.; ANDRADE, W. Investigating test smells in javascript test code. In: *Brazilian Symposium on Systematic and Automated Software Testing*. New York, NY, USA: Association for Computing Machinery, 2021. (SAST'21), p. 36–45.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research*, v. 4, p. 237–285, 1996.

KATZ, D. S.; GRUENPETER, M.; HONEYMAN, T. Taking a fresh look at fair for research software. *Patterns*, v. 2, n. 3, p. 100222, 2021.

KHOMH, F.; VAUCHER, S.; GUÉHÉNEUC, Y.-G.; SAHRAOUI, H. A bayesian approach for the detection of code and design smells. In: *Int.l Conf. on Quality Software*. Jeju, Korea: IEEE, 2009. p. 305–314.

KIM, D. J.; CHEN, T.-H. P.; YANG, J. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, Springer, v. 26, n. 5, p. 1–47, 2021.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. A field study of refactoring challenges and benefits. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2012. (FSE '12).

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, IEEE, v. 40, n. 7, p. 633–649, 2014.

KOOCHAKZADEH, N.; GAROUSI, V. Tecrevis: A tool for test coverage and test redundancy visualization. In: BOTTACI, L.; FRASER, G. (Ed.). *Testing – Practice and Research Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 129–136.

KOOCHAKZADEH, N.; GAROUSI, V. A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering*, Hindawi, v. 2010, 2010.

KRAMER, O. Scikit-learn. In: *Machine learning for evolution strategies*. Switzerland: Springer, 2016. p. 45–53.

KUHN, M.; JOHNSON, K. *Applied predictive modeling*. New York, NY, USA: Springer, 2013.

KUMMER, M.; NIERSTRASZ, O.; LUNGU, M. Categorising test smells. *Bachelor Thesis. University of Bern*, Citeseer, 2015.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUéHéNEUC, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, v. 167, p. 110610, 2020. ISSN 0164-1212.

LAMBIASE, S.; CUPITO, A.; PECORELLI, F.; LUCIA, A. D.; PALOMBA, F. Just-in-time test smell detection and refactoring: The darts project. In: *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: ACM, 2020. p. 441–445.

LAVALLEY, M. P. Logistic regression. *Circulation*, Am Heart Assoc, v. 117, n. 18, p. 2395–2399, 2008.

LIMA, R.; COSTA, K.; SOUZA, J.; TEIXEIRA, L.; FONSECA, B.; D'AMORIM, M.; RIBEIRO, M.; MIRANDA, B. Do you see any problem? on the developers perceptions in test smells detection. In: *Proceedings of the XXII Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '23), p. 21–30. ISBN 9798400707865.

MAIER, F.; FELDERER, M. Detection of test smells with basic language analysis methods and its evaluation. In: IEEE. *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2023. p. 897–904.

MARINKE, R.; GUERRA, E. M.; SILVEIRA, F. F.; AZEVEDO, R. M.; NASCIMENTO, W.; ALMEIDA, R. S. de; DEMBOSCKI, B. R.; SILVA, T. S. da. Towards an extensible architecture for refactoring test code. In: *Computational Science and Its Applications – ICCSA 2019*. Cham: Springer International Publishing, 2019. p. 456–471.

MARTINEZ, M.; ETIEN, A.; DUCASSE, S.; FUHRMAN, C. Rtj: a java framework for detecting and refactoring rotten green test cases. In: IEEE. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA, 2020. p. 69–72.

MARTINS, L.; BEZERRA, C.; COSTA, H.; MACHADO, I. Smart prediction for refactorings in the software test code. In: *Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2021. p. 115–120.

MARTINS, L.; BRITO, V.; FEITOSA, D.; ROCHA, L.; COSTA, H.; MACHADO, I. From blackboard to the office: A look into how practitioners perceive software testing education. In: *Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (EASE 2021), p. 211–220.

MARTINS, L.; CAMPOS, D.; SANTANA, R.; JUNIOR, J.; COSTA, H.; MACHADO, I. Hearing the voice of experts: Unveiling stack exchange communities' knowledge of test smells. In: *IEEE/ACM 16th Int.l Conf. on Cooperative and Human Aspects of Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 80–91.

MARTINS, L.; COSTA, H.; MACHADO, I. On the diffusion of test smells and their relationship with test code quality of java projects. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2532, 2023.

MARTINS, L.; COSTA, H.; RIBEIRO, M.; PALOMBA, F.; MACHADO, I. Automating test-specific refactoring mining: A mixed-method investigation. In: *Proceedings of the 23rd IEEE Int.l Working Conf. on Source Code Analysis and Manipulation*. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 12.

MARTINS, L.; GHALEB, T.; COSTA, H.; MACHADO, I. *Curated dataset of test-specific refactorings*. Figshare, 2023. Available at: <https://figshare.com/s/3cd337c00ba36954854e>.

MARTINS, L.; GHALEB, T.; COSTA, H.; MACHADO, I. *TSR-Catalog: The Catalog of Test Smells Refactorings*. ReadTheDocs, 2023. Available at: <https://tsr-catalog.readthedocs.io/en/latest/>.

MARTINS, L.; PONTILO, V.; COSTA, H.; PALOMBA, F.; MACHADO, I. *Data Collection and analysis for EMSE*. 2023. Accessed on 12.07.2023. Available at: <https://figshare.com/s/2f1d6dc0134f5a95d745>.

MARTINS, L.; PONTILO, V.; COSTA, H.; PALOMBA, F.; MACHADO, I. *Toward Developer-Oriented Test Refactoring Recommendations Through Supervised Machine Learning Algorithm — Online Appendix*. 2023. Available at: <https://figshare.com/s/f1bfa5fdbf1d4caf1d27>.

MCKNIGHT, P. E.; NAJAB, J. Mann-whitney u test. *The Corsini encyclopedia of psychology*, Wiley Online Library, p. 1–1, 2010.

MCMINN, P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 14, n. 2, p. 105–156, 2004.

MELO, S.; MOREIRA, V.; PASCHOAL, L. N.; SOUZA, S. Testing education: A survey on a global scale. In: *In 34th Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2020. p. 554–563.

MENS, T.; TOURWE, T. A survey of software refactoring. *IEEE Transactions on Software Engineering*, v. 30, n. 2, p. 126–139, 2004.

MESZAROS, G. *xUnit test patterns: Refactoring test code*. Upper Saddle River, NJ: Addison-Wesley, 2007. (Addison-Wesley Signature Series).

MESZAROS, G.; SMITH, S. M.; ANDREA, J. The test automation manifesto. In: MAURER, F.; WELLS, D. (Ed.). *Extreme Programming and Agile Methods - XP/Agile Universe 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 73–81.

MICCO, J. The state of continuous integration testing at google. 2017.

MITCHELL, T. M. *Machine learning*. [S.l.]: McGraw-hill, 1997.

MURPHY-HILL, E.; BLACK, A. P. Why don't people use refactoring tools? In: *Proceedings of the 1st Workshop on Refactoring Tools*. [S.l.: s.n.], 2007. p. 61–62.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 1, p. 5–18, 2011.

NELDER, J. A.; WEDDERBURN, R. W. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, Wiley Online Library, v. 135, n. 3, p. 370–384, 1972.

NEMENYI, P. B. *Distribution-free multiple comparisons*. [S.l.]: Princeton University, 1963.

NOBLE, W. S. What is a support vector machine? *Nature biotechnology*, Nature Publishing Group, v. 24, n. 12, p. 1565–1567, 2006.

OPDYKE, W. F. *Refactoring Object-Oriented Frameworks*. PhD Thesis (PhD Thesis) — University of Illinois at Urbana-Champaign, USA, 1992.

O'BRIEN, R. M. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity*, Springer, v. 41, n. 5, p. 673–690, 2007.

PALOMBA, F.; NUCCI, D. D.; PANICHELLA, A.; OLIVETO, R.; LUCIA, A. D. On the diffusion of test smells in automatically generated test code: An empirical study. In: *Proceedings of the 9th International Workshop on Search-Based Software Testing*. New York, NY, USA: ACM, 2016. p. 5–14.

PALOMBA, F.; TAMBURRI, D. A. A.; FONTANA, F. A.; OLIVETO, R.; ZAIDMAN, A.; SEREBRENIK, A. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering*, IEEE, v. 47, p. 108–129, 2018.

PALOMBA, F.; ZAIDMAN, A.; LUCIA, A. D. Automatic test smell detection using information retrieval techniques. In: IEEE. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. New York, NY, USA, 2018. p. 311–322.

PANICHELLA, A.; PANICHELLA, S.; FRASER, G.; SAWANT, A. A.; HELLENDOORN, V. J. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In: IEEE. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. New York, NY, USA, 2020. p. 523–533.

PANICHELLA, A.; PANICHELLA, S.; FRASER, G.; SAWANT, A. A.; HELLENDOORN, V. J. Test smells 20 years later: Detectability, validity, and reliability. *Empirical Software Engineering*, Springer, 2022.

PANTIUCHINA, J.; ZAMPETTI, F.; SCALABRINO, S.; PIANTADOSI, V.; OLIVETO, R.; BAVOTA, G.; PENTA, M. D. Why developers refactor source code: A mining-based study. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, sep 2020. ISSN 1049-331X.

PAULA, E. A. de; BONIFÁCIO, R. Testaxe: Automatically refactoring test smells using junit 5 features. In: SBC. *Anais Estendidos do XIII Congresso Brasileiro de Software: Teoria e Prática*. [S.l.], 2022. p. 89–98.

PECORELLI, F.; CATOLINO, G.; FERRUCCI, F.; LUCIA, A. D.; PALOMBA, F. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In: *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2020. p. 296–307.

PECORELLI, F.; LILLO, G. D.; PALOMBA, F.; LUCIA, A. D. Vitrum: A plug-in for the visualization of test-related metrics. In: *Proceedings of the International Conference on Advanced Visual Interfaces*. New York, NY, USA: Association for Computing Machinery, 2020. (AVI '20).

PECORELLI, F.; PALOMBA, F.; LUCIA, A. D. The relation of test-related factors to software quality: a case study on apache systems. *EMSE*, Springer, v. 26, p. 1–42, 2021.

PENTA, M. D.; BAVOTA, G.; ZAMPETTI, F. On the relationship between refactoring actions and bugs: a differentiated replication. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2020. p. 556–567.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. On the distribution of test smells in open source android applications:

An exploratory study. In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. USA: IBM Corp., 2019. (CASCON '19), p. 193–202.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. Tsdetect: An open source test smells detection tool. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1650–1654.

PERUMA, A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. An exploratory study on the refactoring of unit test files in android applications. In: IEEE. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. New York, NY, USA, 2020. p. 350–357.

PERUMA, A.; SIMMONS, S.; ALOMAR, E. A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, Springer, v. 27, n. 1, p. 1–43, 2022.

PIZZINI, A.; REINEHR, S.; MALUCELLI, A. Automatic refactoring method to remove eager test smell. In: *Proceedings of the XXI Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2023.

PIZZINI, A.; REINEHR, S.; MALUCELLI, A. Sentinel: A process for automatic removing of test smells. In: *Proceedings of the XXII Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '23), p. 80–89. ISBN 9798400707865. Available at: <https://doi.org/10.1145/3629479.3630019>.

QUADRI, S.; FAROOQ, S. U. Software testing–goals, principles, and limitations. *International Journal of Computer Applications*, International Journal of Computer Applications, New York, NY, USA, v. 6, n. 9, p. 1, 2010.

QUINLAN, J. R. Induction of decision trees. *Machine learning*, Springer, v. 1, n. 1, p. 81–106, 1986.

QUINLAN, J. R. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1558602380.

QUSEF, A.; ELISH, M. O.; BINKLEY, D. An exploratory study of the relationship between software test smells and fault-proneness. *IEEE Access*, IEEE, v. 7, p. 139526–139536, 2019.

REICHHART, S.; GÎRBA, T.; DUCASSE, S. Rule-based assessment of test quality. *J. Object Technol.*, Citeseer, v. 6, n. 9, p. 231–251, 2007.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S. Characterizing the relative significance of a test smell. In: *2006 22nd IEEE International Conference on Software Maintenance*. Philadelphia, PA, USA: IEEE, 2006. p. 391–400.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S.; RIEGER, M. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Trans. Softw. Eng.*, IEEE Press, New York, NY, USA, v. 33, n. 12, p. 800–817, dec 2007.

RWEMALIKA, R.; HABCHI, S.; PAPADAKIS, M.; TRAON, Y. L.; BRASSEUR, M.-C. Smells in system user interactive tests. *Empirical Software Engineering*, Springer, v. 28, n. 1, p. 20, 2023.

SAGAR, P. S.; ALOMAR, E. A.; MKAOUER, M. W.; OUNI, A.; NEWMAN, C. D. Comparing commit messages and source code metrics for the prediction refactoring activities. *Algorithms*, MDPI, v. 14, n. 10, p. 289, 2021.

SANTANA, R.; FERNANDES, D.; CAMPOS, D.; SOARES, L.; MACIEL, R.; MACHADO, I. Understanding practitioners' strategies to handle test smells: A multi-method study. In: ______. *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21), p. 49–53.

SANTANA, R.; MARTINS, L.; ROCHA, L.; VIRGINIO, T.; CRUZ, A.; COSTA, H.; MACHADO, I. Raide: A tool for assertion roulette and duplicate assert identification and refactoring. In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 374–379.

SANTANA, R.; MARTINS, L.; VIRGÍNIO, T.; ROCHA, L.; COSTA, H.; MACHADO, I. An empirical evaluation of raide: A semi-automated approach for test smells detection and refactoring. *Science of Computer Programming*, Elsevier, v. 231, p. 103013, 2024.

SANTANA, R.; MARTINS, L.; VIRGíNIO, T.; SOARES, L.; COSTA, H.; MACHADO, I. Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. In: *Anais do XXV Congresso Ibero-Americano em Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2022. p. 263–277.

SARKER, I. H.; KAYES, A.; BADSHA, S.; ALQAHTANI, H.; WATTERS, P.; NG, A. Cybersecurity data science: an overview from machine learning perspective. *Journal of Big Data*, Springer, v. 7, n. 1, p. 1–29, 2020.

SHELDON, M. R.; FILLYAW, M. J.; THOMPSON, W. D. The use and interpretation of the friedman test in the analysis of ordinal-scale data in repeated measures designs. *Physiotherapy Research Int.l*, Wiley Online Library, v. 1, n. 4, p. 221–228, 1996.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: *Proceedings of the 2016 24th ACM SIGSOFT Int.l Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016. p. 858–870.

Silva Junior, N.; MARTINS, L.; ROCHA, L.; COSTA, H.; MACHADO, I. How are test smells treated in the wild? a tale of two empirical studies. *Journal of Software Engineering Research and Development*, v. 9, n. 1, p. 9:1 – 9:16, Sep. 2021.

Silva Junior, N.; SOARES, L. R.; MARTINS, L.; MACHADO, I. A survey on test practitioners' awareness of test smells. *CoRR*, abs/2003.05613, p. 1–14, 2020. Available at: <https://arxiv.org/abs/2003.05613>.

SOARES, E.; ARANDA, M.; OLIVEIRA, N.; RIBEIRO, M.; GHEYI, R.; SOUZA, E.; MACHADO, I.; SANTOS, A.; FONSECA, B.; BONIFÁCIO, R. Manual tests do smell! cataloging and identifying natural language test smells. In: IEEE. *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2023. p. 1–11.

SOARES, E.; RIBEIRO, M.; AMARAL, G.; GHEYI, R.; FERNANDES, L.; GARCIA, A.; FONSECA, B.; SANTOS, A. Refactoring test smells: A perspective from open-source developers. In: *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*. New York, NY, USA: Association for Computing Machinery, 2020. (SAST 20), p. 50–59.

SOARES, E.; RIBEIRO, M.; GHEYI, R.; AMARAL, G.; SANTOS, A. M. Refactoring test smells with junit 5: Why should developers keep up-to-date. *IEEE Transactions on Software Engineering*, p. 1–1, 2022.

SOBRINHO, E. V. de P.; LUCIA, A. D.; MAIA, M. de A. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE TSE*, IEEE, v. 47, n. 1, p. 17–66, 2018.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. Pydriller: Python framework for mining software repositories. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2018. p. 908–911.

SPADINI, D.; PALOMBA, F.; ZAIDMAN, A.; BRUNTINK, M.; BACCHELLI, A. On the relation of test smells to software code quality. In: IEEE. *2018 IEEE international conference on software maintenance and evolution (ICSME)*. New York, NY, USA, 2018. p. 1–12.

SPADINI, D.; SCHVARCBACHER, M.; OPRESCU, A.-M.; BRUNTINK, M.; BACCHELLI, A. Investigating severity thresholds for test smells. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2020. p. 311–321.

STEFANO, M. D.; PECORELLI, F.; NUCCI, D. D.; LUCIA, A. D. A preliminary evaluation on the relationship among architectural and test smells. In: IEEE. *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.], 2022. p. 66–70.

TAHIR, A.; COUNSELL, S.; MACDONELL, S. G. An empirical study into the relationship between class features and test smells. In: IEEE. *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. New York, NY, USA, 2016. p. 137–144.

TANIGUCHI, M.; MATSUMOTO, S.; KUSUMOTO, S. Jtdog: A gradle plugin for dynamic test smell detection. In: *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: IEEE Press, 2021. (ASE '21), p. 1271–1275.

TEIXEIRA, T. S. R.; SILVEIRA, F. F.; GUERRA, E. M. Moving towards a mutant-based testing tool for verifying behavior maintenance in test code refactorings. *Computers*, MDPI, v. 12, n. 11, p. 230, 2023.

TEMPERO, E.; ANSLOW, C.; DIETRICH, J.; HAN, T.; LI, J.; LUMPE, M.; MELTON, H.; NOBLE, J. The qualitas corpus: A curated collection of java code for empirical studies. In: IEEE. *2010 Asia Pacific Software Engineering Conference*. [S.l.], 2010. p. 336–345.

TERRAGNI, V.; SALZA, P.; PEZZE, M. Measuring software testability modulo test quality. In: *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: ACM, 2020. p. 241–251.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, v. 1, n. 1, p. 1–21, 2020.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016. p. 4–15.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, Wiley Online Library, v. 29, n. 4, p. e1838, 2017.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; PENTA, M. D.; LUCIA, A. D.; POSHYVANYK, D. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, v. 43, n. 11, p. 1063–1088, 2017.

VIDAL, S. A.; MARCOS, C.; DÍAZ-PACE, J. A. An approach to prioritize code smells for refactoring. *Automated Software Engg.*, Kluwer Academic Publishers, USA, v. 23, n. 3, p. 501–532, sep 2016. ISSN 0928-8910.

VIRGINIO, T.; MARTINS, L.; ROCHA, L.; SANTANA, R.; CRUZ, A.; COSTA, H.; MACHADO, I. Jnose: Java test smell detector. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2020. p. 564–569.

VIRGINIO, T.; MARTINS, L.; SANTANA, R.; CRUZ, A.; ROCHA, L.; COSTA, H.; MACHADO, I. On the test smells detection: an empirical study on the jnose test accuracy. *Journal of Software Engineering Research and Development*, v. 9, n. 1, p. 8:1 – 8:14, 2021.

VIRGINIO, T.; MARTINS, L. M.; SOARES, L. R.; SANTANA, R.; COSTA, H.; MACHADO, I. An empirical study of automatically-generated tests from the perspective of test smells. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 92–96.

VIRGINIO, T.; SANTANA, R.; MARTINS, L. M.; SOARES, L. R.; COSTA, H.; MACHADO, I. On the influence of test smells on test coverage. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (SBES 2019), p. 467–471.

WANG, T.; GOLUBEV, Y.; SMIRNOV, O.; LI, J.; BRYKSIN, T.; AHMED, I. Pynose: A test smell detector for python. In: *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: IEEE Press, 2021. (ASE '21), p. 593–605.

WEISSGERBER, P.; BIEGEL, B.; DIEHL, S. Making programmers aware of refactorings. In: *WRT*. [S.l.: s.n.], 2007. p. 58–59.

WEISSGERBER, P.; DIEHL, S. Identifying refactorings from source-code changes. In: IEEE. *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. [S.l.], 2006. p. 231–240.

WU, H.; YIN, R.; GAO, J.; HUANG, Z.; HUANG, H. To what extent can code quality be improved by eliminating test smells? In: IEEE. *2022 International Conference on Code Quality (ICCQ)*. New York, NY, USA, 2022. p. 19–26.

YANG, Y.; HU, X.; XIA, X.; YANG, X. The lost world: Characterizing and detecting undiscovered test smells. *ACM Transactions on Software Engineering and Methodology*, ACM New York, NY, 2023.

YEN, S.; LEE, Y. Under-sampling approaches for improving prediction of the minority class in an imbalanced dataset. In: *Intelligent Control and Automation*. Berlin, Heidelberg: Springer, 2006. p. 731–740.

ZEISS, B.; NEUKIRCHEN, H.; GRABOWSKI, J.; EVANS, D.; BAKER, P. Refactoring and metrics for ttcn-3 test suites. In: SPRINGER. *International Workshop on System Analysis and Modeling*. Berlin, Heidelberg, 2006. p. 148–165.

ZHANG, S.; JALALI, D.; WUTTKE, J.; MUSLU, K.; LAM, W.; ERNST, M. D.; NOTKIN, D. Empirically revisiting the test independence assumption. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2014. (ISSTA 2014), p. 385–396.