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

Software development is a collaborative, social, knowledge-intensive activity, and human-centered aspects such as communication and personality can impact software projects. These factors are essential for team diversity. There are three types of team diversity: informational (or knowledge), social, and values diversity. In this context, we have noticed a lot of interest in the software engineering community about the relationship between human factors and code quality. Investigating if knowledge diversity (human aspects) affects test code quality is important. The literature has brought up several studies investigating test code quality. Still, little empirical evidence exists on the effects of knowledge diversity (e.g., educational background, level of professional experience, expertise, and skills) and test code quality. Whether we consider the prevalence of test smells in current software testing research, the number is way more limited. Test smells are bad implementations inserted by developers and can harm the comprehensibility and maintainability of test suites. Recent studies discuss developers' perceptions of test smells and their impact on quality improvement, but there is little evidence regarding the effects of knowledge diversity in this direction, especially concerning interdisciplinary collaboration and the exchange of experiences between teams. This thesis aims to gather empirical evidence on the relationship between developer knowledge diversity and test code quality, particularly the effect test smells can bring on software quality. Initially, we built a knowledge base by considering the main concepts of software testing, maintenance, evolution, test smells, software refactoring, developer classification, and tool support. Next, to accomplish our research objective, we used a mixed-methods approach (e.g., surveys, interviews, and mining of GitHub repositories). We organized the findings as a set of guidelines to support developers in preventing the insertion of test smells during the creation of unit test cases.

Palavras-chave: Software maintenance, software testing, test smells, evidence-based software engineering.

DSC | 52 | 2024

On the Interplay between Developer Knowledge Diversity and Test Code Quality: Understanding and Addressing Test Smells in Software Development

Denivan do Carmo Campos da Silva

UFBA

On the Interplay between Developer Knowledge Diversity and Test Code Quality: Understanding and Addressing Test Smells in Software Development

Denivan do Carmo Campos da Silva

Tese de Doutorado

Universidade Federal da Bahia

Programa de Pós-Graduação em Ciência da Computação

Outubro | 2024

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

ON THE INTERPLAY BETWEEN DEVELOPER' EXPERIENCE AND TEST CODE QUALITY: A PRACTICAL GUIDELINES TO AVOIDING TEST SMELLS IN SOFTWARE DEVELOPMENT

DENIVAN DO CARMO CAMPOS DA SILVA

TESE DE DOUTORADO

Salvador - BA
Outubro de 2024

DENIVAN DO CARMO CAMPOS DA SILVA

ON THE INTERPLAY BETWEEN DEVELOPER' EXPERIENCE AND TEST CODE QUALITY: A PRACTICAL GUIDELINES TO AVOIDING TEST SMELLS IN SOFTWARE DEVELOPMENT

Esta Tese 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

Salvador - BA
Outubro de 2024

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

S586 Silva, Denivan do Carmo Campos da

On the Interplay between Developer' Experience and Test Code Quality: A Practical guidelines to avoiding Test Smells in Software Development. / Denivan do Carmo Campos da Silva. – Salvador, 2024.

108 f.

Orientador: Prof. Dr. Ivan do Carmo Machado.

Tese (Doutorado) – Universidade Federal da Bahia, Instituto de Computação, 2024.

1. Engenharia de software. 2. Manutenção de Software. 3. Teste de Software. 4. Qualidade de software. I. Machado, Ivan do Carmo. II. Universidade Federal da Bahia. III. Título.

CDU:004.41

Termo de Aprovação

Denivan do Carmo Campos da Silva

On the Interplay between Developer Knowledge Diversity and Test Code Quality: Understanding and Addressing Test Smells in Software Development

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

Salvador, 14 de outubro de 2024

gov.br Documento assinado digitalmente
IVAN DO CARMO MACHADO
Data: 16/10/2024 09:41:26-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Ivan do Carmo Machado
(Orientador – PGCOMP/UFBA)

gov.br Documento assinado digitalmente
EDNA DIAS CANEDO
Data: 16/10/2024 09:49:36-0300
Verifique em https://validar.iti.gov.br

Prof.ª Drª. Edna Dias Canedo
(UNB)

gov.br Documento assinado digitalmente
CARLA TACIANA LIMA LOURENCO SILVA SCHUE
Data: 16/10/2024 16:41:38-0300
Verifique em https://validar.iti.gov.br

Prof.ª Drª. Carla Taciana Lima Lourenco Silva Shuenemann
(CIn/UFPE)

gov.br Documento assinado digitalmente
EMANUEL FERREIRA COUTINHO
Data: 16/10/2024 17:38:00-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Emanuel Ferreira Coutinho
(UFC)

gov.br Documento assinado digitalmente
MANOEL GOMES DE MENDONCA NETO
Data: 17/10/2024 08:47:18-0300
Verifique em https://validar.iti.gov.br

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

I dedicate this work first to God, to my family and friends.

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus, a Ele toda honra e toda a Glória! Muito obrigado meu Deus, por tudo e principalmente por sua presença em minha vida! Agradeço a minha família pela força, pelo apoio dado. Agradeço a meu Pai Adaires Dorneles pela força e a preocupação da sua filha estar longe e dando forças para que meu sonho se ralizasse.

Quero agradecer ao meu orientador Ivan Machado, que durante esses cinco anos me orientou. Obrigada pelo acolhimento e pelas oportunidades concedidas, as experiências eu pude ter e pelas suas correções que foram dadas, pois me ensinaram a ser uma pessoa melhor. Agradeço a todos os professores do Programa de Pós-Graduação em Computação da Universidade Federal da Bahia, o PGCOMP/UFBA, que de uma forma ou de outra me ensiram o caminho a ser seguido.

Agradeço à Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB), pelo apoio financeiro. Sem a bolsa de estudos não seria possível realizar este trabalho. Quero agradecer também a ACM-Women Scholarship, uma iniciativa da ACM para apoiar a participação feminina em eventos da área de Computação. Tal apoio foi fundamental para o custeio de viagens para apresentação de artigos. Como destaque, o apoio financeiro desta iniciativa possibilitou a minha participação na ICSE 2023, evento realizado em Melbourne, Austrália.

Não poderia deixar de agradecer aos funcionários do CEAPG-IC/UFBA pela atenção, disponibilidade e presteza ímpares, ao longo desses cinco anos. Meu muito obrigada à vocês Samuel, Mayara, Kleber e Miquéias.

Agradeço ao nosso grupo de pesquisa ARIES LAB, pelo tempo que passamos juntos e contribuímos com os trabalhos uns dos outros. Obrigada Professor Ivan por liderar esse grupo de pesquisa, e nos orientar. Obrigada Luana Martins, Railana Santana, Sara Lima pelas parcerias no desenvolvimento dos artigos. Além disso, agradeço à professora Carla Bezerra e seu orientando Humberto Damasceno pela parceria.

Quero agradecer em especial aos meus amigos *"mais chegados que irmão"*, que durante o doutorado estiveram comigo: Edilton Lima, Sávio Emanuel, Luana Martins, Renata Souza, Railana Santana. Agradeço-lhes pela força, pelas risadas e por nos apoiar como irmãos nas horas difíceis. Agradeço também aos professores e alunos do Laboratório Inteligência Artificial Aplicada (LIAA), onde passei o meu último ano. Foi um prazer trabalhar com vocês, e obrigada pelo apoio e compreensão.

Agradeço às meninas que dividiram apartamento comigo em Salvador, em especial à Luana Martins que, além de colega de apartamento, minha companheira de pesquias e viagens, que juntas propiciamos um ambiente familiar e divertido durante esses anos anos. Meu muito obrigada!

ACKNOWLEDGEMENTS

Firstly, I thank God, and to Him, all honor and glory! Thank you very much, my God, for everything, especially your presence in my life! I thank my family for the strength and the support given. I thank my Father, Adaires Dorneles, for the strength and concern of his daughter being far away and giving strength for my dream to come true.

I want to thank my advisor, Ivan Machado, who guided me during these five years. Thank you for the welcome and the opportunities granted, the experiences I could have and your corrections, which taught me to be a better person. I thank all the professors of the Postgraduate Program in Computing at the Federal University of Bahia, PGCOMP/UFBA, who, in one way or another, showed me the way to follow.

I thank the *Fundação de Amparo à Pesquisa do Estado da Bahia* (FAPESB) for the financial support. Without the scholarship, it would not be possible to carry out this work. I also want to thank ACM-Women Scholarship, an ACM initiative to support female participation in Computing events. Such support was crucial for covering travel expenses for paper presentations. As a highlight, the financial support from this initiative enabled my participation in ICSE 2023, an event held in Melbourne, Australia.

I couldn't fail to thank the employees of CEAPG-IC/UFBA for their unique attention, availability, and promptness over these five years—my heartfelt thanks to you, Samuel, Mayara, Kleber, and Miquéias.

I thank our research group ARIES LAB, for the time we spent together and contributed to each other's work. Thank you, Professor Ivan, for leading this research group and guiding us. Thank you, Luana Martins, Railana Santana, and Sara Lima, for the partnerships in developing the research papers. Additionally, I thank Professor Carla Bezerra and her student, Humberto Damasceno, for the partnership.

I want to especially thank my friends, "closer than brothers," who were with me during my Ph.D.: Edilton Lima, Sávio Emanuel, Luana Martins, Renata Souza, and Railana Santana. I thank them for their strength and laughs and for supporting us like brothers in challenging times. I also thank the Applied Artificial Intelligence Laboratory (LIAA) professors and students, where I spent my last year. It was a pleasure to work with you. Thank you for your support and understanding.

I thank the girls who shared an apartment with me in Salvador, especially Luana Martins, who, besides being a roommate, was my research and travel companion, and together we provided a familiar and fun environment during these years. Thank you very much!

"The ultimate measure of a man is not where he stands in moments of comfort and convenience, but where he stands at times of challenge and controversy"
Martin Luther King Jr.

RESUMO

Projetos de software open-source resultam de contribuições de pessoas com todos os níveis de habilidade. O desenvolvimento de software é uma atividade colaborativa, social e intensiva em conhecimento, e aspectos centrados no ser humano, como comunicação e personalidade, podem impactar projetos de software. Esses fatores são essenciais para a diversidade da equipe. Existem três tipos de diversidade de equipe: diversidade informacional (ou de conhecimento), social e de valores. Nesse contexto, temos notado muito interesse na comunidade de Engenharia de Software sobre a relação entre fatores humanos e a qualidade do código. Investigar se a diversidade de conhecimento (aspectos humanos) afeta a qualidade do código de teste é importante. A literatura apresenta vários estudos que investigam a qualidade do código de teste. Ainda assim, existem poucas evidências empíricas sobre os efeitos da diversidade de conhecimento (por exemplo, formação educacional, nível de experiência profissional, expertise e habilidades) e a qualidade do código de teste. Se considerarmos a prevalência de "test smells" na pesquisa atual de testes de software, o número é ainda mais limitado. Test smells são implementações ruins inseridas por desenvolvedores e podem prejudicar a compreensibilidade e a manutenibilidade dos conjuntos de testes. Estudos recentes discutem as percepções dos desenvolvedores sobre test smells e seu impacto na melhoria da qualidade, mas há pouca evidência acerca dos efeitos da diversidade de conhecimento nessa direção, especialmente no que diz respeito à colaboração interdisciplinar e à troca de experiências entre equipes. Neste contexto, esta tese tem como objetivo reunir evidências empíricas sobre a relação entre a diversidade de conhecimento dos desenvolvedores e a qualidade do código de teste, particularmente o efeito que os test smells podem trazer na qualidade do software. Inicialmente, construímos uma base de conhecimento considerando os principais conceitos de testes de software, manutenção, evolução, test smells, refatoração de software, classificação de desenvolvedores e suporte ferramental. Em seguida, para alcançar nosso objetivo de pesquisa, utilizamos uma abordagem de métodos mistos, que englobou o planejamento e a execução de *surveys*, entrevistas e mineração de repositórios do GitHub. Organizamos as descobertas como um conjunto de diretrizes para apoiar os desenvolvedores na prevenção da inserção de test smells durante a criação de casos de teste de unidade.

Palavras-chave: Manutenção de software, testes de software, test smells, engenharia de software baseada em evidências.

ABSTRACT

Software development is a collaborative, social, knowledge-intensive activity, and human-centered aspects such as communication and personality can impact software projects. These factors are essential for team diversity. There are three types of team diversity: informational (or knowledge), social, and values diversity. In this context, we have noticed a lot of interest in the software engineering community about the relationship between human factors and code quality. Investigating if knowledge diversity (human aspects) affects test code quality is important. The literature has brought up several studies investigating test code quality. Still, little empirical evidence exists on the effects of knowledge diversity (e.g., educational background, level of professional experience, expertise, and skills) and test code quality. Whether we consider the prevalence of test smells in current software testing research, the number is way more limited. Test smells are bad implementations inserted by developers and can harm the comprehensibility and maintainability of test suites. Recent studies discuss developers' perceptions of test smells and their impact on quality improvement, but there is little evidence regarding the effects of knowledge diversity in this direction, especially concerning interdisciplinary collaboration and the exchange of experiences between teams. This thesis aims to gather empirical evidence on the relationship between developer knowledge diversity and test code quality, particularly the effect test smells can bring on software quality. Initially, we built a knowledge base by considering the main concepts of software testing, maintenance, evolution, test smells, software refactoring, developer classification, and tool support. Next, to accomplish our research objective, we used a mixed-methods approach (e.g., surveys, interviews, and mining of GitHub repositories). We organized the findings as a set of guidelines to support developers in preventing the insertion of test smells during the creation of unit test cases.

Keywords: Software maintenance, software testing, test smells, evidence-based software engineering.

CONTENTS

List of Figures xv

List of Tables xvi

List of Acronyms xvii

Chapter 1—Introduction 1

1.1 Context and Motivation 1
1.2 Problem Statement 3
1.3 Research Objective 3
1.3.1 Specific Objectives 4
1.3.2 Research Questions 4
1.4 Methodology 6

Chapter 2—Background 8

2.1 Software Testing 8
2.1.1 Testing Techniques 9
2.2 Software Maintenance and Evolution 9
2.3 Software refactoring 10
2.4 Test Smells 11
2.5 Developers' experience classification 25
2.6 Software Testing Tools 26
2.6.1 JNose Test Tool 26
2.6.2 SmartGit Tool 27
2.6.3 Seart Tool 29

Chapter 3—Related work 30

3.1 Classifying Developers' Experience Levels 30
3.2 Understanding Developers' perception of test smells 31
3.3 Exploring Developers' Experience and Refactoring 32
3.4 Survivability of Test Smells 33
3.5 Developers' Perceptions Through Stack Exchange Topics 35

Chapter 4—Developers' perception on the severity of test smells 37

4.1 Introduction 37
4.2 Research Methodology 38
4.2.1 Study Design 38
4.3 Results 40
4.3.1 Developers' Profiles 40
4.3.2 Characterization of the selected projects 41
4.3.3 How do software developers consider the severity of test smells and their effects on test code quality? (RQ1) 41
4.3.4 What is the behavior of test code after refactoring test smells? (RQ2) 43
4.4 Discussion 45
4.5 Threats to Validity 46
4.6 Conclusion 47

Chapter 5—Understanding practitioners' strategies to handle test smells 48

5.1 Introduction 48
5.2 Research Methodology 49
5.3 Survey 50
5.3.1 Survey methodology 50
5.3.2 Survey results 52
5.3.2.1 Test code creation and maintenance (RQ1) 52
5.3.2.2 Awareness about test smells (RQ2) 53
5.3.2.3 Severity of test smells (RQ_3) 55
5.3.2.4 How test smells are handled (RQ_4) 55
5.4 Interview 56
5.4.1 Interview methodology 56
5.4.2 Interview results 57
5.4.2.1 Test code creation and maintenance (RQ_1) 58
5.4.2.2 Awareness about test smells (RQ_2) 58
5.4.2.3 Severity of test smells (RQ_3) 60
5.4.2.4 How test smells are handled (RQ_4) 61
5.4.2.5 Quality improvement strategies (RQ_5) 62
5.5 Discussion 63
5.5.1 Test code creation and maintenance (RQ_1) 63
5.5.2 Awareness about test smells (RQ_2) 64
5.5.3 Severity of test smells (RQ_3) 65
5.5.4 How test smells are handled (RQ_4) 66
5.5.5 Quality improvement strategies (RQ_5) 66
5.6 Threats to Validity 67
5.7 Conclusion 68

Chapter 6—On the influence of developers' experience on software test code quality 69

6.1 Introduction . . . 69
6.2 Research Methodology . . . 70
6.2.1 Classification of core and peripheral developers . . . 72
6.2.2 Data Collection . . . 73
6.3 Results . . . 74
6.3.1 Developer's experience and test case quality (RQ1) . . . 74
6.3.2 Relationship between test smell insertion/ removal and developer's experience (RQ2) . . . 77
6.3.3 Moment when developers to insert or remove test smells (RQ3) . 79
6.4 Discussion . . . 84
6.5 Threats to Validity . . . 86
6.6 Conclusion . . . 86

Chapter 7—Developer Contributions to Test Smell Survivability: A Study of OSS Projects 88

7.1 Introduction . . . 88
7.2 Research Methodology . . . 89
7.2.1 Study Design . . . 89
7.2.2 Classification of core and peripheral developers . . . 91
7.2.3 Data Collection . . . 91
7.3 Results . . . 93
7.3.1 Lifetime of test smells in software projects (RQ1) . . . 93
7.3.2 Impact of developer experience on the survivability of test smells (RQ2) . . . 96
7.4 Discussion . . . 100
7.4.1 Lifetime of test smells in software projects (RQ1) . . . 100
7.4.2 Impact of developer experience on the survivability of test smells (RQ2) . . . 101
7.5 Threats to Validity . . . 101
7.6 Conclusion . . . 102

Chapter 8—Test Smells Knowledge in Stack Exchange Communities 104

8.1 Introduction . . . 104
8.2 Research methodology . . . 105
8.2.1 Identification of discussions . . . 106
8.2.2 Classification of discussions . . . 109
8.2.3 Data Analysis . . . 109
8.3 Results . . . 111
8.3.1 Discussions characterization . . . 111
8.3.2 Trends and Challenges (RQ1) . . . 112

8.3.3 Test code problems (RQ2) 115
8.3.4 Test code refactorings (RQ3) 117
8.3.5 Test behavior (RQ4) 120
8.4 Discussion 120
8.5 Threats to Validity 122
8.6 Conclusion 122

Chapter 9—Test Smell Prevention Guidelines 124

9.1 Test Smells Prevention Guidelines 124
9.1.1 Test code 124
9.1.2 Test smells prevention 125
9.1.2.1 Assertion Roulette (AR) 125
9.1.2.2 Conditional Test Logic (CTL) 126
9.1.2.3 Duplicate Assert (DA) 126
9.1.2.4 Resource Optimism (RO) 127
9.1.2.5 Sleepy Test (ST) 128
9.1.2.6 Unknown Test (UT) 129
9.2 Empirical Evaluation 130
9.2.1 Research Questions 130
9.2.2 Empirical Evaluation Overview 131
9.2.3 Participants' characterization 133
9.3 Results 135
9.3.1 Usefulness of the proposed guidelines (RQ1) 135
9.3.2 Main challenges faced when developing the test cases (RQ2) . . . 136
9.4 Threats to Validity 140
9.5 Conclusion 140

Chapter 10—Conclusions 142

10.1 Open Issues 143
10.2 Impacts of this thesis 143
10.3 Threats to validity of the thesis 144
10.4 Scientific Contributions 145

Appendix A—Survey and interview studies 160

A.1 Survey on test quality practices 160
A.1.1 Experience with the JUnit framework 160
A.1.2 Demographic information and participant experience 160
A.1.3 Developers' perspective on creating and maintaining tests 160
A.1.4 Developers' perspective on test quality 161
A.1.5 Developers' perspective on Test Smells 169
A.1.6 Acknowledgement 170

Appendix B—Guideline Evaluation 171

B.1 Experimental Design 171
B.2 Study on test quality practices 172
B.2.1 Informed Consent Form 172
B.2.2 Experience with the JUnit framework 172
B.2.3 Participants' Characterization 172
B.2.4 Guideline Evaluation 175
B.2.5 Feedback from the evaluation of the study on test quality practices 177

LIST OF FIGURES

1.1 Overview of the research methodology. 6

2.1 The “onion model.” Adapted by Terceiro, Rios and Chavez (2010) from Crowston and Howison (2005). 26
2.2 JNose Evolution overview 27
2.3 Example of the history of commits visualized wit *SmartGit Tool* 28
2.4 Seart Tool overview 29

4.1 Empirical Study Design 39

6.1 Study Design 71
6.2 Number of commits in test classes of each project by developers’ role . . 75
6.3 Number of test smells inserted by developers’ role 76
6.4 Number of insertions and removals by developers 78
6.5 Number of test smells inserted by developers 78
6.6 Number of test smells removed by developers 79
6.7 Insertion of test smells when deleting the test class, deleting test methods, or refactorings. 79
6.8 Removal of test smells when deleting the test class, deleting test methods or refactorings. 80
6.9 Evolution of the *code4craft/webmagic* project 81
6.10 Evolution of the *hs-web/hsweb-framework* project 82
6.11 Evolution of the *airbnb/lottie-android* project 83
6.12 Evolution of the *apolloconfig/apollo* project 84

7.1 Study Design 90
7.2 Classification of the longevity of test smells 94
7.3 Longevity and remained of test smells 94
7.4 The test smells regarding removal in software projects 95
7.5 Survivability of test smells concerning the removal of test smells in the project 96

8.1 Study Design 106
8.2 Sample question post extracted from StackOverflow. 107
8.3 Accepted/useful answer from a StackOverflow post. 108
8.4 Time between asking a question and receiving a first answer. 111
8.5 Time between asking a question and closing the discussion. 111
8.6 Classification of types of questions and challenges 112

8.7 Classification of GD and SD on test smells according to Garousi's catalog of test smells (GAROUSI; KÜÇÜK, 2018). 116
8.8 Suggestions of patterns for code organization, good practices, and test code refactorings for preventing and fixing test smells. 118

9.1 Empirical evaluation overview 130
9.2 Developers' knowledge of test smells 134
9.3 Distribution of treatments and tasks 134
9.4 Perceived usefulness (U), ease of use (E), and self-predicted future use (S) of the test smell prevention guidelines. The information is showed using a six-point Likert scale, ranging from "Extremely unlikely" (1) to "Extremely likely" (6) 136

B.1 Diagram 1 - Relationship between the class MathOperations and IOHandler175

LIST OF TABLES

4.1 Selected projects . . . 41
4.2 Degree of Severity of Test Smell per Project . . . 42
4.3 Maintainability and impact of test smells per project . . . 43

5.1 Key topics of research questions addressed in the survey and interview. . . 50
5.2 The most popular test smells . . . 54
5.3 Participants' opinion whether test smells affect the test . . . 54
5.4 Frequency that test smells are inserted . . . 55
5.5 Level of severity of test smells . . . 55
5.6 Opinion on the need to refactor test smells . . . 56
5.7 Frequency of test smells refactoring . . . 56
5.8 Profile of interview participants . . . 58

6.1 Summary of selected projects . . . 75
6.2 Test Smells Detected . . . 77

7.1 *hsweb/hsweb-framework* projects . . . 97
7.2 *apolloconfig/apollo* projects . . . 98
7.3 *airbnb/lottie-android* projects . . . 98
7.4 *code4craft/webmagic* projects . . . 98
7.5 All projects . . . 100

8.1 Groups of tags composing the search string. . . . 108
8.2 Number or retrieved discussions. . . . 109
8.3 Data items extracted from discussions. . . . 110

9.1 List of tasks the developers have completed . . . 132
9.2 Questionnaire used to assess the factors U, E and S. . . . 135
9.3 Factor Analysis . . . 137

B.1 Treatment with guideline . . . 171
B.2 Treatment without guideline . . . 172
B.3 Utilization (U) of the guidelines . . . 178
B.4 Use (E) of the guidelines . . . 178
B.5 Use (S) of the guidelines . . . 178

LIST OF ACRONYMS

General acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
GUI	Graphical User Interface
IDE	Integrated Development Environment
LoC	Lines of Code
OSS	Open-Source Software
PUT	Product Over Testing
SQA	Software Quality Assurance
UML	Unified Modeling Language

Test smells acronyms

AR	Assertion Roulette
CTL	Conditional Test Logic
CI	Constructor Initialization
DA	Duplicate Assert
ET	Eager Test
EpT	Empty Test
ECT	Exception Catching Throwing
EH	Exception Handlin
GF	General Fixture
IT	Ignored Test
LT	Lazy Test

MNT	Magic Number Test
MG	Mystery Guest
PS	Print Statement
RA	Redundant Assertion
RO	Resource Optimism
SE	Sensitive Equality
ST	Sleepy Test
TRW	Test Run War
VT	Verbose Test
UT	Unknown Test

Chapter 1

INTRODUCTION

This chapter serves as an introduction to our research on test smells in test code. In Section 1.1, "Context and Motivation", we provide the necessary background and rationale behind our study, highlighting the significance of addressing test smells in ensuring the quality and reliability of software systems. Section 1.2, "Problem Statement", explores the specific challenges posed by test smells in test code, identifying the key issues our research aims to tackle. In Section 1.3, "Research Objective", we outline the specific goals and objectives of our study, detailing the outcomes we aim to achieve. Finally, in Section 1.4, "Methodology", we describe the approach and methods we employed to conduct our research, including data collection, analysis, and validation strategies.

1.1 CONTEXT AND MOTIVATION

Software development is a collaborative, social, and knowledge-intensive activity that involves human-centered aspects (e.g., communication and personality). These aspects can affect the development schedule, the quality of software projects, and team productivity (PINZGER; NAGAPPAN; MURPHY, 2008; FAGERHOLM; MÜNCH, 2012; DEMARCO; LISTER, 2013; TAMBURRI *et al.*, 2015; FREIRE *et al.*, 2021).

We have observed a growing interest in the software engineering community to explore the relationship between human aspects and software quality. Empirical evidence claims that (i) code ownership, (ii) developer experience, and (iii) organizational structure can significantly impact software quality (HERBSLEB; GRINTER, 1999; BOH; SLAUGHTER; ESPINOSA, 2007; PINZGER; NAGAPPAN; MURPHY, 2008; RAHMAN; DEVANBU, 2011; ALOMAR *et al.*, 2020):

(i) Code ownership aims to simplify decision-making for managers in organizations by identifying those responsible for authoring code and finding specialists to whom tasks should be delegated. Some researchers have studied the effects of code ownership, as we sample next.

Bird *et al.* (2010) claim that source code ownership is a key issue for large-scale software development. In such a study, the authors discover that major contributors, or those with significant ownership, have a weak effect on quality, particularly in environments lacking formal policies. Conversely, the presence of minor contributors shows a strong correlation with software failures, notably in the Windows Vista operating system project (focus of their research). However, this relationship is less pronounced in Eclipse and Firefox navigators. The report concludes that metrics like the number of low-experienced developers and the ownership ratio to the primary owner are significant indicators of pre-release and post-release defects.

Rahman and Devanbu (2011) discussed the influence of code ownership on software quality. The authors observed that in large projects, a file may either be solely managed by one developer or worked on by multiple developers. Through an empirical evaluation, they could find a correlation between the number of developers working on a file and the incidence of defects. Additionally, they analyzed the impact of individual developers' contributions to modules or files containing bugs. Despite being conducted a decade ago, this study's empirical evidence remains relevant and widely discussed today.

(ii) Developer experience can be associated with various aspects such as awareness of the programming language, level of education, and project contributions, to name a few. The more a developer contributes to a project, the higher her expertise (BOH; SLAUGHTER; ESPINOSA, 2007).

In addition to discussing the effect of code ownership, Rahman and Devanbu (2011) also examined the influence of developers' experience on software quality. The authors investigated whether the experience of multiple developers and their collaborations on the same file relate to defects. They also explore whether individual developer contributions to defective files affect the overall quality of the software.

AlOmar *et al.* (2020) conducted an exploratory study on different levels of knowledge (e.g., experience score). The authors investigated whether developers with greater experience are more likely to be responsible for many refactoring activities or if only subsets of developers perform the main refactoring activities.

(iii) Organizational structure can be associated with activities such as task allocation, coordination, and supervision. Nagappan, Murphy and Basili (2008) investigated the relationship between organizational structure and software quality. The authors performed an empirical evaluation of the effectiveness of organizational metrics, such as the organizational distance of developers, the number of developers working on a component, the number of tasks developers perform in the organization, and the number of changes to a component within the context of that organization, to identify faults in the analyzed project. The study focused on developer experience and the quality of test code to identify the occurrence of failures, defects, and refactoring in projects.

The literature also includes studies investigating test case refactoring with test smells and how this refactoring affects software quality (MARTINS *et al.*, 2023a). Considering the studies on developer experience and test code quality, there is a growing concern with refactoring and detecting test smells. Our motivation is understanding how to avoid inserting test smells when creating test code.

1.2 PROBLEM STATEMENT

Test smells are bad implementations developers insert that can harm test suites' comprehensibility and maintainability (DEURSEN *et al.*, 2001; BAVOTA *et al.*, 2015; GAROUSI; KÜÇÜK, 2018; ALJEDAANI *et al.*, 2021; CAMPOS; ROCHA; MACHADO, 2021). Analogous to production code, test code can also encompass defects and anti-patterns, which might affect its quality. Additionally, test cases with test smells tend to be less effective in detecting defects (ATHANASIOU *et al.*, 2014).

Consequently, test smells can make maintenance more challenging and compromise test effectiveness (GAROUSI; KÜÇÜK, 2018). Aligned with the community's increasing focus on understanding how human factors impact software quality and addressing test smells through detection and refactoring strategies, we propose to investigate the intersection of these areas.

1.3 RESEARCH OBJECTIVE

This thesis aims to analyze human factors' impact on test code quality, particularly concerning test smells, and to develop effective strategies for mitigating their occurrence.

We seek to gather empirical evidence on how developers' experience with open-source software (OSS) projects influences the introduction and refactoring of test smells. Additionally, it aims to uncover the underlying causes behind the emergence and elimination of test smells over time. By addressing these objectives, the study contributes to a proper understanding of how human factors may impact test code quality. Its outcome may be helpful for software practitioners to propose effective strategies for managing and improving software testing practices in OSS environments.

1.3.1 Specific Objectives

The following specific objectives have been defined for this study:

O1. Investigate developers' perceptions of test smells' effects on test quality and how they manage these issues.

O2. Examine how developers' experience, particularly in different project contexts, influences the introduction and removal of test smells.

O3. Analyze the lifecycle of test smells during software maintenance and evolution.

O4. Assess the prevalence and density of test smells across OSS projects.

O5. Identify the challenges developers face when addressing test smells and analyze the refactoring strategies they apply in practice.

1.3.2 Research Questions

The primary research question guiding this study is:

How do human factors and technical practices influence the prevention and survivability of test smells in software development?

From this question, we derived the following specific research questions (RQ):

RQ1. How do software developers consider the severity of test smells and their effects on test code quality? This question investigates the impact of test smells on comprehensibility and maintainability from the viewpoint of developers involved in open-source projects hosted on GitHub.

RQ2. What is the practitioners' perception about the test smells effects on maintainability and test code quality? This question aims to study how the practitioners identify the need for refactoring test sets, the frequency of such activities, and the methodologies used to address test smells within their projects.

RQ3. How does the developer's experience in OSS projects affect the quality of test cases? This question studies the correlation between developers' experience and the occurrence of test smells in four OSS projects (*airbnb/lottie-android*, *code4craft/web-magic*, *hs-web/hsweb-framework*, and *apolloconfig/apollo*). It aims to determine whether more experienced developers contribute to fewer test smells or engage more actively in refactoring.

RQ4. What is the lifetime of test smells in software projects? This question investigates the prevalence of recurring test smells—those frequently introduced and removed—in the same four open-source projects (from RQ3). Additionally, it examines whether collaborative efforts to refactor test cases result in concurrently removing associated test smells.

RQ5. What is the behavior of test code after refactoring test smells? This question explores patterns in creating and removing test smells during the lifecycle of software projects. It seeks to identify whether refactoring activities are concentrated among specific developers and whether these activities reduce the overall number of test smells.

RQ6. What challenges do developers report for handling problems in the test code? This question categorizes challenges reported by developers in Stack Exchange communities regarding test code refactoring. It analyzes questions posed by developers to understand common obstacles and classifies them into categories (why-how-what) to highlight specific difficulties.

RQ7. What preventive and corrective actions do developers suggest to handle test smells in the test code? This question aims to identify proactive and reactive strategies suggested by developers for mitigating test smell occurrences. It focuses on commonly recommended refactoring techniques employed to fix test smells.

The contributions of this study are outlined as follows:

- Understanding developers' perceptions of test smells, particularly regarding their impact on test quality and software maintainability.
- Analyzing the experiences and practices of core and peripheral developers in managing test smells during software maintenance and evolution.
- Investigating the challenges developers face when addressing test smells.
- The empirical studies provide a diagnosis regarding the developer's experience and test code quality.
- Creating guideline support to aid software testing practitioners in proactively avoiding introducing test smells in their projects.

1.4 METHODOLOGY

Our study methodology consists of three phases, as shown in Figure 1.1, which are detailed next.

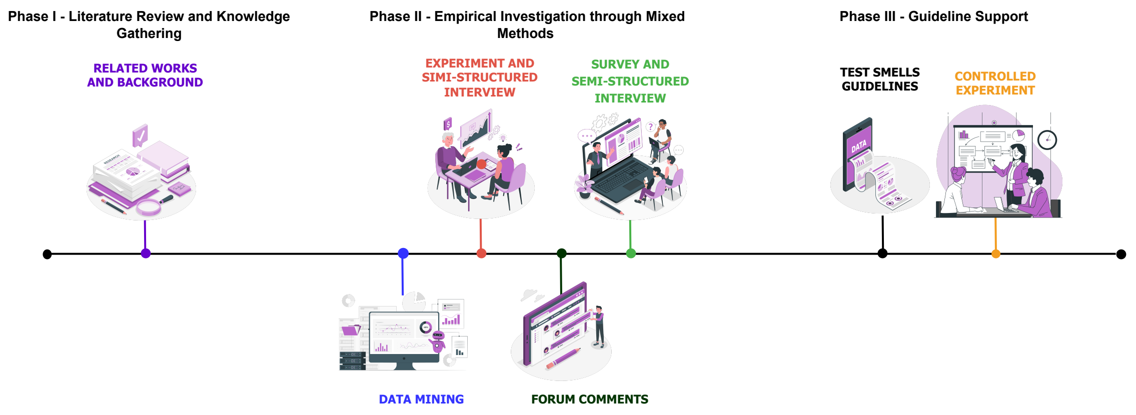

Phase I - Literature Review and Knowledge Gathering
RELATED WORKS AND BACKGROUND
Phase II - Empirical Investigation through Mixed Methods
EXPERIMENT AND SIMI-STRUCTURED INTERVIEW
SURVEY AND SEMI-STRUCTURED INTERVIEW
Phase III - Guideline Support
TEST SMELLS GUIDELINES
CONTROLLED EXPERIMENT
DATA MINING
FORUM COMMENTS

Figure 1.1: Overview of the research methodology.

- **Phase I - Literature Review and Knowledge Gathering:** This initial phase is conceptual and addresses the key concepts relevant to our research. We built the body of knowledge in an *ad-hoc* manner, focusing on essential software testing concepts, software maintenance and evolution, test smells, and testing tools. We relied on leading venues (conferences and journals) in the Software Engineering field to establish the foundational concepts of our work.

- **Phase II - Empirical Investigation through Mixed Methods:** We employed a combination of research methods, including surveys, interviews, and data mining, to understand the diversity of developers' knowledge and its impact on test code quality (HESSE-BIBER, 2010). As detailed below, each research method was used to address specific research questions in this thesis.

  - **Empirical Study (RQ1 and RQ5):** We combined data mining approaches and interviews with developers managing a set of OSS projects (WOHLIN *et al.*, 2012). We investigated how developers perceive the severity and impact of test smells in the test code they write.
  - **Surveys and Interviews (RQ2):** We employed surveys and interviews (KASUNIC, 2005; WOHLIN *et al.*, 2012), combining qualitative and quantitative approaches to gather information (YIN, 2017). This study explored test creation and maintenance strategies using developers' perceptions of eight test smells.
  - **Empirical Study (RQ3):** By employing a data mining approach (KHLEEL; NEHÉZ, 2020), we conducted an empirical study to analyze the relationship between developer experience and test code quality. This study examines the insertion and removal of test smells and code ownership in OSS projects.
  - **Empirical Study (RQ4):** We employed a data mining approach (KHLEEL; NEHÉZ, 2020) to understand the correlation between developer experience and the survivability of test smells, along with collecting developers' suggestions for refactorings in OSS projects.
  - **Empirical Study (RQ6 and RQ7):** In this last study, we employed a data mining approach to conduct a qualitative analysis (WOHLIN *et al.*, 2012) of discussions on the StackExchange network,[1] identifying challenges and refactoring suggestions for unit tests.

- **Phase III - Guideline Support:** In this phase, we analyze and interpret the data collected in Phase II, focusing on identifying trends, challenges, and the impact of developer experience on test smells. Based on such observations, we designed and empirically evaluated the perceived usefulness and perceived ease-of-use of the proposed guidelines.

[1]⟨https://stackexchange.com/⟩

Chapter 2

BACKGROUND

This chapter introduces the underlying concepts relevant to this thesis. It encompasses the following sections: Section 2.1 introduces fundamental software testing concepts. Section 2.2 introduces software maintenance and evolution. Section 2.3 covers software refactoring. Section 2.4 discusses test smells. Section 2.5 presents how the literature addresses developers' experience classification. Finally, Section 2.6 introduces the tools we used to support our research.

2.1 SOFTWARE TESTING

Software testing has been demonstrated to be the key approach for evaluating software quality in a real-world environment (AMMANN; OFFUTT, 2016). Software testing aims to identify the faults in the Product Over Testing (PUT), source code defects, and prescribed documentation during the project's initial phase (GUERRA; COLOMBO, 2009; SPÍNOLA *et al.*, 2019).

The literature presents several definitions for software testing. It can be defined as "the process that generates evidence of discovering defects in software systems" (POHL; BOCKLE; LINDEN, 2005), for instance. According to Jalote (2012), it is necessary to execute a set of test cases to test a project. Test cases comprise input values, execution conditions, and expected results for a given test. Test case executions can be positive (or valid) or negative (or invalid). That is, test cases with positive results confirm that the software does what it is supposed to do, while tests with negative results are those in the product different from what was expected.

Throughout this section, we discuss the application of the software testing process and some essential concepts and terminologies.

2.1.1 Testing Techniques

The literature presents different testing techniques. According to Luo (2001), Delamaro, Jino and Maldonado (2013), the techniques highlight the quality aspects of software systems. Two widely accepted testing techniques exist: *functional* and *structural.* These are briefly introduced next.

- **Functional testing**. It is a test technique that aims to test only the external behavior of the software without the need to test its structure (code). Functional testing is also known as ***black-box*** testing. The purpose of black-box testing is to ensure the quality of creating data input sets (test cases) to validate the conforming output of the system or component as per specified program functional requirements.
- **Structural testing**. It is a testing technique that tests the implemented code and ensures the quality of the internal structure of the software. It is also known as ***white-box*** testing. White-box testing aims to select the specific test cases to be executed in the software as specific instructions, branches, or paths of the program (PRESSMAN; MAXIM, 2020). White-box testing is particularly important as it explores the logical paths of the program where logic errors can occur. Moreover, structural testing can be applied at different stages during the software development process.

2.2 SOFTWARE MAINTENANCE AND EVOLUTION

As society's dependence on software systems has grown over time (MENS *et al.*, 2010; MENS; SEREBRENIK; CLEVE, 2014), the need for software to adapt to changing requirements and functions has become crucial. This ongoing adaptation and enhancement process is referred to as *software maintenance.*

According to the IEEE (1983), software maintenance refers to modifying a software product after its delivery to fix post-release faults, enhance performance, and tailor the product to a new environment (TORCHIANO; RICCA; LUCIA, 2007). Likewise, as earlier noted by Canning (1972), practitioners often interpret maintenance as encompassing error correction, expansion, extending software functionality, and adapting to changes in software or hardware.

Researchers and practitioners have also proposed to use the term "software evolution" as a viable alternative to "maintenance" (BENNETT; RAJLICH, 2000; TRIPATHY; NAIK, 2014). Software evolution has faced an increased significance and is now a first-class citizen for software developers to consider in their projects (BENNETT; RAJLICH, 2000).

Lehman (1980), Lehman and Ramil (2003), Tripathy and Naik (2014) claimed that software evolution is deemed essential for maintaining the utility of software over time. Lehman and Ramil (2000), Rajlich (2014) also argued that software evolution encompasses all programming activities, iterative changes based on requirements, and the adoption of new technologies, all aimed at creating a new version of software from an existing operational version.

2.3 SOFTWARE REFACTORING

Software refactoring involves changing the structure of source code without affecting its functionality or its observable behavior (FOWLER, 1999; ALIZADEH *et al.*, 2020; SALGADO, 2020). This practice aims to promote better design and coding habits through incremental code transformations (SANTANA *et al.*, 2021), improving the codebase's readability and maintainability (DIBBLE; GESTWICKI, 2014). By improving code readability and maintainability, refactoring also aids in bug detection, making it a routine practice in modern software engineering (JACOBSON *et al.*, 2019; JUNG *et al.*, 2019). Refactoring becomes particularly important in large systems where multiple developers collaborate without a comprehensive understanding of the entire system (CHRISTOPOULOU *et al.*, 2012).

The refactoring process typically consists of four key phases (KATAOKA *et al.*, 2002; KATIĆ; FERTALJ, 2009): *(i)* identifying candidates for refactoring, *(ii)* selecting the appropriate refactoring technique, *(iii)* applying the refactoring, and *(iv)* validating its effect. Identifying refactoring opportunities requires an in-depth knowledge of the system and an understanding of best coding practices (OLIVEIRA *et al.*, 2020). Since detecting issues in the code can be a time-consuming and complex task, developers often rely on automated tools to facilitate the process (PAIVA *et al.*, 2017). Moreover, choosing the best refactoring technique for each situation can present further challenges due to the complexity of different systems (TEMPERO; GORSCHEK; ANGELIS, 2017). Comprehensive strategies for refactoring both production and test code are detailed in the works of Fowler (1999) and Deursen *et al.* (2001), respectively.

2.4 TEST SMELLS

Test smells are indications of potential issues or deficiencies in unit test code, often resulting from suboptimal design or implementation choices made by developers (PALOMBA *et al.*, 2016; PERUMA, 2018; GAROUSI; KÜÇÜK, 2018; ALJEDAANI *et al.*, 2021).

The literature discusses various types of test smells. Deursen *et al.* (2001) introduced a catalog with eleven types and strategies to refactor them. Later, Garousi and Küçük (2018) expanded this catalog to include an additional 80 types of test smells. Additionally, Peruma (2018) introduced 12 new types, inspired by common bad practices encountered during test programming.

Other researchers have investigated the impact of test smells on test code (MESZAROS; SMITH; ANDREA, 2003; MESZAROS, 2007; SPADINI *et al.*, 2018; PERUMA *et al.*, 2019; VIRGÍNIO *et al.*, 2019). Test smells found in test code can serve as indicators of low software quality, leading to decreased code comprehension and negatively affecting test code maintainability (GREILER; DEURSEN; STOREY, 2013). In the following section, we introduce the specific test smells that are considered in this thesis.

- **Assertion Roulette (AR).** This smell occurs when a test method contains lots of assertion statements without description/message or parameter in the assertion method (DEURSEN *et al.*, 2001).

  Detection: A test method that contains multiple assertion statements without an argument as a parameter.

  Example: Listing 2.1 presents a test method containing *AR* (line 14). The example presents a method of test class `AuthenticationTests` of the `hsweb-framework`.[1]

[1]Available at ⟨https://github.com/hs-web/hsweb-framework⟩ project

```java
public class AuthenticationTests {

    @Test
    public void testInitUserRoleAndPermission() {
        Authentication authentication = builder.user("{\"id\":\"admin\",\"username\":\"admin\",\"name\":\"
            ↪ Administrator\",\"userType\":\"default\"}")
                .role("[{\"id\":\"admin-role\",\"name\":\"admin\"}]")

                .permission("[{\"id\":\"user-manager\",\"actions\":[\"query\",\"get\",\"update\"]" +
                        ",\"dataAccesses\":[{\"action\":\"query\",\"field\":\"test\",\"fields
                             ↪ \":[\"1\",\"2\",\"3\"],\"scopeType\":\"CUSTOM_SCOPE\",\"type\":\"DENY_FIELDS
                             ↪ \"}]}]")

                .build();

        //test user
        assertEquals(authentication.getUser().getId(), "admin");
        assertEquals(authentication.getUser().getUsername(), "admin");
        assertEquals(authentication.getUser().getName(), "Administrator");
        assertEquals(authentication.getUser().getUserType(), "default");
 }
```

Listing 2.1: Example - *Assertion Roulette*

- **Constructor Initialization (CI).** It occurs when a test method implements a constructor. It is recommended that all the fields be initialized inside the *setUp()* method (PERUMA, 2018).

  Detection: The setUp () method needs to initialize the fields.

  Example: Listing 2.2 presents a test method containing *CI* (line from 2 to 24). the example presents a method of test class `BetterImageSpanTest` from the `fresco` project.

- **Conditional Test Logic (CTL).** This smell occurs when the test has conditional or repetitive structures (PERUMA, 2018).

  Detection: A test method that contains one or more control statements, i.e, if, switch, conditional expression, for, foreach, or while statements.

  Example: Listing 2.3 presents a test method containing *CTL* (lines from 8 to 10). The example presents a method of test class `FastBeanCopierTest` from the `hsweb-framework` project.

```java
public class BetterImageSpanTest {
  public BetterImageSpanTest(
      String description,
      int alignment,
      int drawableHeight,
      int fontAscent,
      int fontDescent,
      int expectedAscent,
      int expectedDescent,
      int fontTop,
      int fontBottom,
      int expectedTop,
      int expectedBottom) {
    mDescription = description;
    mAlignment = alignment;
    mDrawableHeight = drawableHeight;
    mFontAscent = fontAscent;
    mFontDescent = fontDescent;
    mExpectedAscent = expectedAscent;
    mExpectedDescent = expectedDescent;
    mFontTop = fontTop;
    mFontBottom = fontBottom;
    mExpectedTop = expectedTop;
    mExpectedBottom = expectedBottom;
  }
```

Listing 2.2: Example - *Constructor Initialization*

- **Duplicate Assert (DA).** This smell occurs when a test case tests the same condition multiple times, which may increase test overhead (PERUMA, 2018).

  Detection: In the test method, there is more than one assertion statement with the same parameters.

  Example: Listing 2.4 presents a test method containing *DA* (lines from 9 to 13). The example presents a method of test class `HashMapTwoFactorTokenManagerTest` from the `hsweb-framework` project.

- **Exception Catching Throwing (ECT).** This smell occurs when the result of the test method depends on the production method throwing an exception (PERUMA, 2018).

  Detection: Test method depends on the production method throwing an exception.

  Example: Listing 2.5 presents a test method containing *ECT* (lines from 8 to 14). The example presents a method of test class `UserTokenManagerTests` from the `hsweb-framework` project.

- **Exception Handlin (EH).** It occurs when the approval or disapproval of a test method explicitly depends on the production method that throws an exception

```java
public class FastBeanCopierTest {
    @Test
    public void testProxy() {
        AtomicReference<Object> reference=new AtomicReference<>();

        ProxyTest test = (ProxyTest) Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),
                new Class[]{ProxyTest.class}, (proxy, method, args) -> {
                    if (method.getName().equals("getName")) {
                        return "test";
                    }

                    if (method.getName().equals("setName")) {
                        reference.set(args[0]);
                        return null;
                    }

                    return null;
                });
    }
}
```

Listing 2.3: Example - *Conditional Test Logic*

```java
public class HashMapTwoFactorTokenManagerTest {
    HashMapTwoFactorTokenManager tokenManager = new HashMapTwoFactorTokenManager();

    @Test
    @SneakyThrows
    public void test() {
        TwoFactorToken twoFactorToken = tokenManager.getToken("test", "test");

        Assert.assertTrue(twoFactorToken.expired());
        twoFactorToken.generate(1000L);
        Assert.assertFalse(twoFactorToken.expired());
        Thread.sleep(1100);
        Assert.assertTrue(twoFactorToken.expired());
    }
}
```

Listing 2.4: Example - *Duplicate Assert*

(try/catch) (PERUMA *et al.*, 2019).

Detection: Checks if the method has any try/catch structures.

Example: Listing 2.6 presents a test method containing *EH* (lines from 8 to 4). he example presents a method of test class `UserTokenManagerTests` from the `hsweb-framework` project.

```java
public class UserTokenManagerTests {
@Test
    public void testDeny() throws InterruptedException {
        DefaultUserTokenManager userTokenManager = new DefaultUserTokenManager();
        userTokenManager.setAllopatricLoginMode(AllopatricLoginMode.deny);
        userTokenManager.signIn("test", "sessionId", "admin", 10000).subscribe();

        try {
            userTokenManager.signIn("test2", "sessionId", "admin", 30000).block();
            Assert.assertTrue(false);
        } catch (AccessDenyException e) {

        }
    }
```

Listing 2.5: Example - *Exception Catch/Throw*

```java
public class UserTokenManagerTests {
@Test
    public void testDeny() throws InterruptedException {
        DefaultUserTokenManager userTokenManager = new DefaultUserTokenManager();
        userTokenManager.setAllopatricLoginMode(AllopatricLoginMode.deny);
        userTokenManager.signIn("test", "sessionId", "admin", 10000).subscribe();

        try {
            userTokenManager.signIn("test2", "sessionId", "admin", 30000).block();
            Assert.assertTrue(false);
        } catch (AccessDenyException e) {

        }
    }
```

Listing 2.6: Example - *Exception Handling*

- **Empty Test (EpT).** This smell occurs when a test method does not contain a single executable instruction (PERUMA, 2018).

  Detection: An empty test method, i.e., does not contain a single executable instruction.

  Example: Listing 2.7 presents a test method containing *EpT* (lines from 2 to 5). The example presents a method of test class `StringUtilsTest` of the `mybatis-plus` project.[2]

- **Eager Test (ET).** This smell occurs when a test method contains multiple calls to multiple production methods (DEURSEN *et al.*, 2001).

  Detection: A test method contains multiple calls to multiple production methods.

  Example: Listing 2.8 presents a test method containing *ET* (lines from 8 to 12). The example presents a method of test class `RedisUserTokenManagerTest` of the

[2]Available at ⟨https://github.com/baomidou/mybatis-plus⟩

```java
class StringUtilsTest {
    @Test
    void canBeAColumnName() {
// assertTrue(StringUtils.canBeColumnName("a$"));
    }
```

Listing 2.7: Example - *Empty Test*

```java
public class RedisUserTokenManagerTest {

@Test
    public void testAuth() {
        Authentication authentication = new SimpleAuthentication();
        tokenManager.signIn("testAuth", "test", "test", 1000, authentication)
                    .as(StepVerifier::create)
                    .expectNextMatches(token -> token.getAuthentication() == authentication)
                    .verifyComplete();
        tokenManager.getByToken("testAuth")
                    .cast(AuthenticationUserToken.class)
                    .as(StepVerifier::create)
                    .expectNextMatches(token -> token.getAuthentication() != null)
                    .verifyComplete();
    }
}
```

Listing 2.8: Example - *Eager Test*

`hsweb-framework` project.

```
1  public class ForwardingRequestListenerTest {
2    @Before
3    public void setUp() {
4      MockitoAnnotations.initMocks(this);
5      mRequestListener1 = mock(RequestListener.class);
6      mRequestListener2 = mock(RequestListener.class);
7      mRequestListener3 = mock(RequestListener.class);
8      when(mRequestListener1.requiresExtraMap(mRequestId)).thenReturn(false);
9      when(mRequestListener2.requiresExtraMap(mRequestId)).thenReturn(false);
10     when(mRequestListener3.requiresExtraMap(mRequestId)).thenReturn(false);
11     mListenerManager =
12         new ForwardingRequestListener(
13             Sets.newHashSet(mRequestListener1, mRequestListener2, mRequestListener3));
14   }
15 }
```

Listing 2.9: Example - *General Fixture*

- **General Fixture (GF).** This smell occurs when a developer writes a *setUp()* method to be used by all test methods in the same test class (DEURSEN *et al.*, 2001).

  Detection: The fields instantiated in the setUp method are not used by all test methods in the same test class.

  Example: Listing 2.9 presents a test method containing *GF* (lines 5 and 7). The example presents a method of test class `ForwardingRequestListenerTest` from the `fresco` project.[3]

- **Ignored Test (IT).** This smell occurs when a method contains the *@Ignore* annotation (PERUMA, 2018).

  Detection: The class or test method contains the *@Ignore* annotation.

  Example: Listing 2.10 presents a test method with *IT* (lines from 2 to 27). The example presents a method of test class `LottieTaskTest` from the `lottie-android` project.[4].

- **Lazy Test (LT).** This smell occurs when multiple test methods check the same (production code) class method (DEURSEN *et al.*, 2001).

  Detection: Various test methods call the same production method.

  Example: Listing 2.11 presents a test method containing *LT* (line from 7 to 20). The example presents a method of test class `DefaultBasicAuthorizeDefinitionTest` from the `hsweb-framework` project.

[3]Available at ⟨https://github.com/facebook/fresco⟩
[4]Available at ⟨https://github.com/airbnb/lottie-android.git⟩

```java
public class LottieTaskTest extends BaseTest {
  @Ignore
  @Test
  public void testRemoveListener() {
    final Semaphore lock = new Semaphore(0);
    LottieTask<Integer> task = new LottieTask<>(new Callable<LottieResult<Integer>>() {
      @Override public LottieResult<Integer> call() {
        return new LottieResult<>(5);
      }
    })
        .addListener(successListener)
        .addFailureListener(failureListener)
        .addListener(new LottieListener<Integer>() {
          @Override public void onResult(Integer result) {
            lock.release();
          }
        });
    task.removeListener(successListener);
    try {
      lock.acquire();
    } catch (InterruptedException e) {
      throw new IllegalStateException(e);
    }
    verifyZeroInteractions(successListener);
    verifyZeroInteractions(failureListener);
  }
}
```

Listing 2.10: Example - *Ignore Test*

- **Mystery Guest (MG).** This smell occurs when a test method utilizes external resources (e.g. files, database, etc) (DEURSEN *et al.*, 2001).

  Detection: A test method containing object instances of files and database classes.

  Example: Listing 2.12 presents a test method containing *MG* (line from 2 to 13). The example presents a method of test class `FileUtilsTest` from the `fresco` project.

- **Magic Number Test (MNT).** This smell occurs when a line with an assertion method contains a magic number as an argument. Magic numbers have no meaning or purpose and should be replaced by constants or variables, thus providing a valid argument to the assertion statement (PERUMA, 2018).

  Detection: The magic number is used as an argument for the assertion method.

  Example: Listing 2.13 presents a test method containing *MNT* (line 9). The example presents a method of test class `EntityEventListenerTest` from the `hsweb-framework` project.

```java
public class DefaultBasicAuthorizeDefinitionTest {
@Test
    @SneakyThrows
    public void testCustomAnn() {
        AopAuthorizeDefinition definition =
                DefaultBasicAuthorizeDefinition.from(TestController.class, TestController.class.getMethod("
                     ↪ test"));
        ResourceDefinition resource = definition.getResources()
                .getResource("test").orElseThrow(NullPointerException::new);
        Assert.assertNotNull(resource);
        Assert.assertTrue(resource.hasAction(Arrays.asList("add")));
        Assert.assertTrue(resource.getAction("add")
                .map(act->act.getDataAccess().getType("user_own_data"))
                .isPresent());
    }
    @Test
    @SneakyThrows
    public void testNoMerge() {
        AopAuthorizeDefinition definition =
                DefaultBasicAuthorizeDefinition.from(TestController.class, TestController.class.getMethod("
                     ↪ noMerge"));
        Assert.assertTrue(definition.getResources().isEmpty());
    }
```

Listing 2.11: Example - *Lazy test*

```java
public class FileUtilsTest {
  @Test
  public void testMkDirsSuccessfulCreate() {
    File directory = mock(File.class);
    when(directory.exists()).thenReturn(false);
    when(directory.mkdirs()).thenReturn(true);
    when(directory.isDirectory()).thenReturn(true);
    try {
      FileUtils.mkdirs(directory);
    } catch (FileUtils.CreateDirectoryException cde) {
      fail();
    }
  }
```

Listing 2.12: Example - *Mystery Guest*

- **Print Statement (PS).** Print statements in unit tests are redundant as unit tests are executed as part of an automated script and do not affect the failing or passing of test cases. Furthermore, they can increase execution time if the developer calls a long-running method from within the print method (i.e., as a parameter) (KIM; CHEN; YANG, 2021).

  Detection: A line invokes the instructions of methods like *print()* or *println()* or *printf()*.

  Example: Listing 2.14 presents a test method containing *PS* (line 19). The example presents a method of test class `DiffTest` from the `hsweb-framework` project.

```java
public class EntityEventListenerTest {
    @Test
    public void test() {
        Mono.just(EventTestEntity.of("test", 1))
            .as(reactiveRepository::insert)
            .as(StepVerifier::create)
            .expectNext(1)
            .verifyComplete();
        Assert.assertEquals(listener.created.getAndSet(0), 1);

    }
}
```

Listing 2.13: Example - *Magic Number Test*

```java
public class DiffTest {
@Test
    public void mapTest() {
        Map<String, Object> before = new HashMap<>();
        before.put("name", "name");
        before.put("age",21);
        before.put("bool", true);
        before.put("bool", false);
        before.put("birthday", DateFormatter.fromString("19910101"));

        Map<String, Object> after = new HashMap<>();
        after.put("name", "name");
        after.put("age", "21");
        after.put("bool", "true");
        after.put("bool", "false");
        after.put("birthday", "1991-01-01");

        List<Diff> diffs = Diff.of(before, after);
        System.out.println(diffs);
        Assert.assertTrue(diffs.isEmpty());
    }
}
```

Listing 2.14: Example - *Print Statement*

- **Redundant Assertion (RA).** This smell occurs when the test methods contain assertion statements that are always true or always false (PERUMA, 2018).

  Detection: A test method contains an assertion statement in which the expected and actual parameters are the same.

  Example: Listing 2.15 presents a test method containing *RA* (line 9). The example presents a method of test class `UserTokenManagerTests` from the `hsweb-framework` project.

- **Resource Optimism (RO).** This smell occurs when a test method uses an instance of a File class without calling the object's *exists()*, *isFile()* or *noExists()* methods (DEURSEN *et al.*, 2001).

  Detection: Test method contains an instance of a File class without calling the

```java
public class UserTokenManagerTests {
 @Test
   public void testDeny() throws InterruptedException {
       DefaultUserTokenManager userTokenManager = new DefaultUserTokenManager();
       userTokenManager.setAllopatricLoginMode(AllopatricLoginMode.deny);
       userTokenManager.signIn("test", "sessionId", "admin", 10000).subscribe();
       try {
           userTokenManager.signIn("test2", "sessionId", "admin", 30000).block();
           Assert.assertTrue(false);
       } catch (AccessDenyException e) {
       }
       Assert.assertTrue(userTokenManager.getByToken("test").block().isNormal());
       Assert.assertNull(userTokenManager.getByToken("test2").block());
   }
}
```

Listing 2.15: Example - *Redundant Assertion*

```java
1  public class FileUtilsTest {
2    @Test
3    public void testRenameSuccessful() {
4      File sourceFile = mock(File.class);
5      File targetFile = mock(File.class);
6
7      when(sourceFile.renameTo(targetFile)).thenReturn(true);
8
9      try {
10       FileUtils.rename(sourceFile, targetFile);
11     } catch (FileUtils.RenameException re) {
12       fail();
13     }
14   }
15 }
```

Listing 2.16: Example - *Resource Optimism*

methods *exists()*, *isFile()* or *noExists()* methods of the object.

Example: Listing 2.16 presents a test method containing *RO* (Lines from 2 to 15). The example presents a method of test class `FileUtilsTest` from the `fresco` project.

- **Sensitive Equality (SE).** This smell occurs when the test method invokes the *toString()* method of an object (DEURSEN *et al.*, 2001).

  Detection: The test method invokes the toString() method of an object.

  Example: Listing 2.17 presents a test method containing *SE* (line 8). The example presents a method of test class `CompareUtilsTest` from the `hsweb-framework` project.

- **Sleepy Test (ST).** This smell occurs when a line that invokes the *Threadsleep()* method (PERUMA, 2018).

```java
public class CompareUtilsTest {
    @Test
    public void dateTest() {

        Date date = new Date();

        Assert.assertTrue(CompareUtils.compare(date, new Date(date.getTime())));
        Assert.assertTrue(CompareUtils.compare(date, DateFormatter.toString(date, "yyyy-MM-dd")));
        Assert.assertTrue(CompareUtils.compare(date, DateFormatter.toString(date, "yyyy-MM-dd HH:mm:ss")))
            ↪ ;
        Assert.assertTrue(CompareUtils.compare(date, date.getTime()));
        Assert.assertTrue(CompareUtils.compare(date.getTime(), date));
    }
```

Listing 2.17: Example - *Sensitive Equality*

```java
1  public class HashMapTwoFactorTokenManagerTest {
2      HashMapTwoFactorTokenManager tokenManager = new HashMapTwoFactorTokenManager();
3
4      @Test
5      @SneakyThrows
6      public void test() {
7          TwoFactorToken twoFactorToken = tokenManager.getToken("test", "test");
8
9          Assert.assertTrue(twoFactorToken.expired());
10         twoFactorToken.generate(1000L);
11         Assert.assertFalse(twoFactorToken.expired());
12         Thread.sleep(1100);
13         Assert.assertTrue(twoFactorToken.expired());
14     }
15 }
```

Listing 2.18: Example - *Sleepy Test*

Detection: A test method that invokes the *Thread.sleep()* method.

Example: Listing 2.18 presents a test method containing *ST* (line 12). The example presents a method of test class `HashMapTwoFactorTokenManagerTest` from the `hsweb-framework` project.

- **Test Run War (TRW).** It occurs when test method allocates resources (e.g., tmp files) also used by other test methods (DEURSEN *et al.*, 2001).

  Detection: Check whether the test method makes use of external resources and whether these resources are used by other test methods.

  Example: Listing 2.19 presents a test method containing *TRW* (lines from 5 to 13). The example presents a method of test class `testReadImageFromCache` and `testDeleteImageFromCache` which uses the same resource *(product.png)*.

```
@test
public class testReadImageFromCache() {
    Image expectedImage = new Image(readBitmapfromAssets(PRODUCT_IMG);

    Image image = cache.readImage("product.png");

    assetsEquals("Read image from cache", expectedImage, image);
}

@test
public testDeleteImageFromCache() {

    boolean deleteSucess = cache.deleteImage("product.png");

    assertTrue(deleteSucess);
}
```

Listing 2.19: Example - *Test Run War*

```
 1  public class RedisUserTokenManagerTest {
 2  @Test
 3      @SneakyThrows
 4      public void testDeny() {
 5          tokenManager.signIn("test-token_offline3", "deny", "user2", 1000)
 6                      .map(UserToken::getToken)
 7                      .as(StepVerifier::create)
 8                      .expectNext("test-token_offline3")
 9                      .verifyComplete();
10          tokenManager.signIn("test-token_offline4", "deny", "user2", 1000)
11                      .map(UserToken::getToken)
12                      .as(StepVerifier::create)
13                      .expectError(AccessDenyException.class)
14                      .verify();
15      }
```

Listing 2.20: Example - *Unknown Test*

- **Unknown Test (UT).** This test smell occurs when a test method does not contain a single assertion and @Test(expected) annotation parameter. An assertion statement describes an expected condition for a test method (PERUMA, 2018).

  Detection: A test method does not contain a single assertion statement and **@Test** (expected) annotation parameter.

  Example: Listing 2.20 presents a test method containing *UT* (line from 2 to 15). The example presents a method of test class `RedisUserTokenManagerTest` from the `hsweb-framework` project.

```java
 1  public class GenericReactiveCacheSupportCrudServiceTest {
 2
 3      @Autowired
 4      private TestCacheEntityService entityService;
 5
 6      @Test
 7      public void test() {
 8
 9          TestEntity entity = TestEntity.of("test2",100);
10
11          entityService.insert(Mono.just(entity))
12                  .as(StepVerifier::create)
13                  .expectNext(1)
14                  .verifyComplete();
15
16          entityService.findById(Mono.just(entity.getId()))
17                  .map(TestEntity::getId)
18                  .as(StepVerifier::create)
19                  .expectNext(entity.getId())
20                  .verifyComplete();
21
22          entityService.getCache()
23                  .getMono("id:".concat(entity.getId()))
24                  .map(TestEntity::getId)
25                  .as(StepVerifier::create)
26                  .expectNext(entity.getId())
27                  .verifyComplete();
28
29          entityService.createUpdate()
30                  .set("age",120)
31                  .where("id",entity.getId())
32                  .execute()
33                  .as(StepVerifier::create)
34                  .expectNext(1)
35                  .verifyComplete();
36
37          entityService.getCache()
38                  .getMono("id:".concat(entity.getId()))
39                  .switchIfEmpty(Mono.error(NullPointerException::new))
40                  .as(StepVerifier::create)
41                  .expectError(NullPointerException.class)
42                  .verify();
43      }
```

Listing 2.21: Example - *Verbose Test*

- **Verbose Test (VT).** This test smell occurs when the tests use too much code to do what they are supposed to do. In other words, the test code is not clean and simple (VIRGíNIO *et al.*, 2021).

  Detection: Method with more than 30 lines including non-executable statements and annotations.

  Example: Listing 2.21 presents a test method containing *VT* (lines from 6 to 43). The example presents a method of test class `GenericReactiveCacheSupport CrudServiceTest` from the `hsweb-framework` project.

2.5 DEVELOPERS' EXPERIENCE CLASSIFICATION

In the software industry, projects often involve teams of individuals collaborating on software development tasks. Recently, there has been a growing interest in researching human aspects due to their impact on project effectiveness. Factors such as organizational structures (NAGAPPAN; MURPHY; BASILI, 2008), code ownership (PINZGER; NAGAPPAN; MURPHY, 2008), and developer experience (BOH; SLAUGHTER; ESPINOSA, 2007) have been shown to influence software quality significantly (RAHMAN; DEVANBU, 2011). Coordinated handling of these concerns can aid managers in improving quality outcomes and decision-making processes.

The literature on developers' experience can be categorized into two main areas. The first area focuses on prior developer experience, encompassing factors such as "years of experience", "professional skills", and "education". These aspects often measure project productivity, maintenance effort, and other quality attributes (BANKER; DATAR; KEMERER, 1991; DARCY *et al.*, 2005; BEAVER; SCHIAVONE, 2006). The second area involves developer experience within the context of specific projects (TERCEIRO; RIOS; CHAVEZ, 2010).

Our study focuses on the latter: the developer's experience within the project. This aspect reflects the developer's role and involvement within a specific project (MOCKUS; FIELDING; HERBSLEB, 2002; CROWSTON; HOWISON, 2005).

According to Matsumoto *et al.* (2010), Zhou and Mockus (2010), the developer's experience within the project context can be measured by analyzing commit information in version control systems. This analysis includes factors such as the number of commits made by the developer over time (measured in days, from the first commit made by the developer in the project to the final commit).

Mockus, Fielding and Herbsleb (2002) claim that a developer's participation in a project is determined by the number of contributions she makes, which is then assessed to understand the decision-making authority they hold within the project. The onion model, as introduced in (CROWSTON; HOWISON, 2005) and depicted in Figure 2.1, illustrates the experience of developers in an Open-Source Software (OSS) project based on their contributions.

For this thesis, we adopted the strategies used by Terceiro, Rios and Chavez (2010) to introduce a classification system for developers' experience within the project, distinguishing between ***core*** and ***peripheral*** developers based on their commit activity.

We establish a threshold at the 80% percentile of commit counts and categorize developers with commit counts above this threshold as ***core*** developers. Those with commit

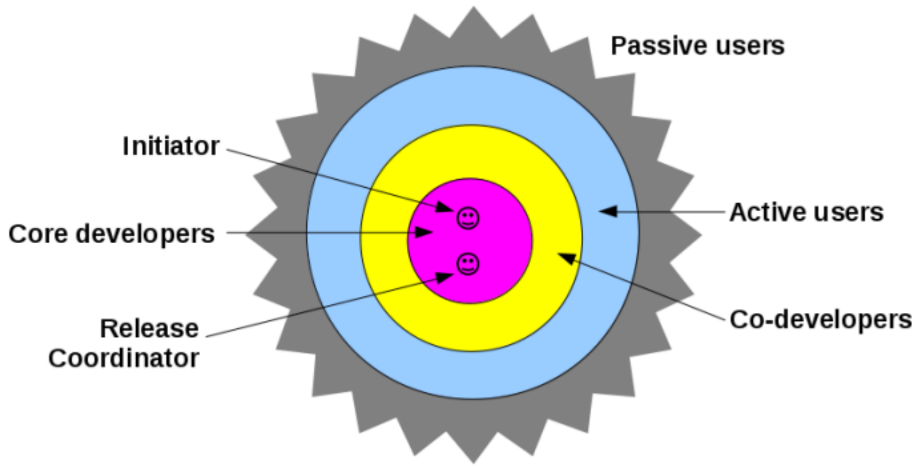


Figure 2.1: The "onion model." Adapted by Terceiro, Rios and Chavez (2010) from Crowston and Howison (2005).

counts below the threshold are classified as ***peripheral*** developers (TERCEIRO; RIOS; CHAVEZ, 2010; BIRD *et al.*, 2010; CALEFATO *et al.*, 2022).

Smaller groups within the project are typically comprised of ***core*** or *central* developers, as these individuals handle the majority of the workload (TERCEIRO; RIOS; CHAVEZ, 2010). On the other hand, the larger group of developers, referred to as ***peripheral*** developers, contribute more directly but less frequently. These developers are often involved in tasks such as defect fixes, documentation, and other minor tasks (TERCEIRO; RIOS; CHAVEZ, 2010).

Core developers typically take on a larger share of project activities and hold greater decision-making authority, whereas ***peripheral*** developers are involved in a smaller subset of activities and have less influence over project decisions. Additionally, ***core*** and ***peripheral*** developers have distinct behaviors when it comes to engaging in discussions about the project (TERCEIRO; RIOS; CHAVEZ, 2010).

2.6 SOFTWARE TESTING TOOLS

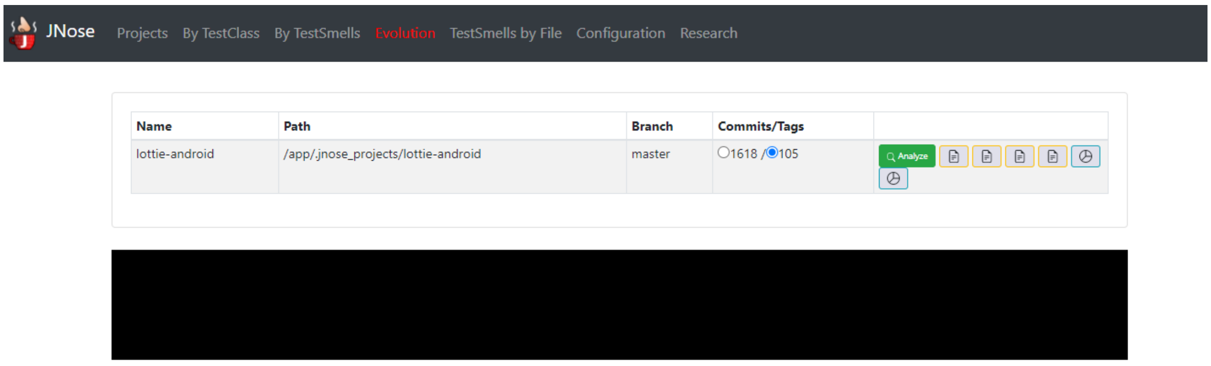
This section introduces the support tools used in our empirical studies.

2.6.1 JNose Test Tool

In a recently published literature review (ALJEDAANI *et al.*, 2021), the authors identified over twenty tools designed specifically for detecting test smells. Most of the tools support the JUnit framework and Java programming language. For our study, we applied the *JNose Test* (VIRGÍNIO *et al.*, 2020a), as it elaborates on the state-of-the-art tool by providing users with an easy-to-use graphical interface that facilitates the detection of test smells. It analyzes Java projects that use Maven and JUnit Framework.[5]

JNose Test is a tool that has the purpose of detecting test smells and calculating test metrics, and one is a web-based application (VIRGÍNIO *et al.*, 2020a).[6] It detects 21 types of test smells by following the detection rules from the *TsDetect tool* (PERUMA *et al.*, 2020).

The tool is composed of rules that identify and quantify the test smell types in each test class and support the analysis of test smells across several project versions (VIRGÍNIO *et al.*, 2020a, 2020b). Once started, the user must configure data entry to enable and specify one of the four types of analysis mode *TestClass*, *TestSmell*, *TestFile*, and *Evolution*. The results shown by *JNose Test* comprise the number of Lines of Code (LoC), methods, types, and the number of test smells in each test class in the project. *JNose Test* can analyze the test quality from an evolutionary perspective, capturing metrics and the occurrence of test smells across multiple versions of a given project. Figure 2.2 shows the feature Handling evolution within the *JNose Test*.



Name	Path	Branch	Commits/Tags	
lottie-android	/app/.jnose_projects/lottie-android	master	○1618 /●105	Analyze

Figure 2.2: JNose Evolution overview

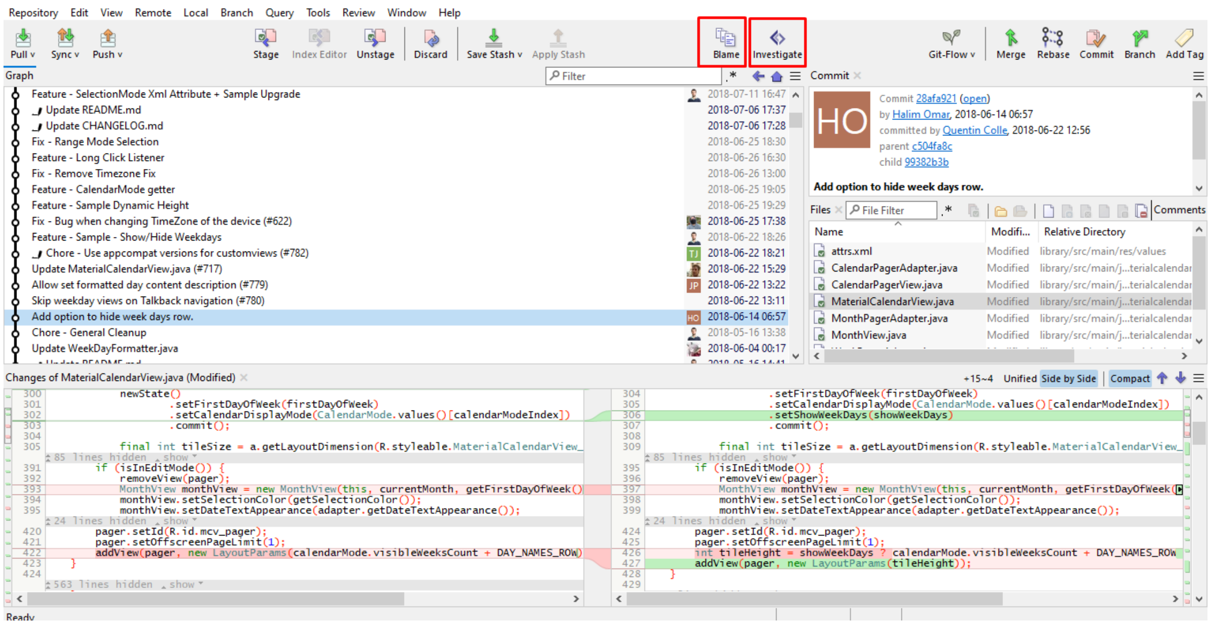
2.6.2 SmartGit Tool

SmartGit is a tool that allows managing repositories and working with Git through a graphical interface.[7] It is a Git client for Windows, Mac, and Linux (MARSH; BEL-

[5]⟨https://junit.org/⟩
[6]⟨https://github.com/arieslab/jnose⟩
[7]⟨https://www.syntevo.com/smartgit/⟩

GUITH; DARGAHI, 2019) operating systems. *SmartGit* is offered in two versions: a commercial and a non-commercial (free, but limited) version. The tool has been used by over 100,000 developers, being the most popular Git Graphical User Interface (GUI) client in the world.

SmartGit has a graphical interface, and in the Log files view, it can select the files of a commit. In the *Graph* part, it is possible to see the project's merges, branches, and pull requests. The features *Blame* and *investigate* support changes as a unified *diff* that can show long lines of *diff* side by side. In addition, all modes support syntax coloring and preparing or deleting changes, as Figure 2.3 shows.



Feature - SelectionMode Xml Attribute + Sample Upgrade	2018-07-11 16:47
Update README.md	2018-07-06 17:37
Update CHANGELOG.md	2018-07-06 17:28
Fix - Range Mode Selection	2018-06-25 18:30
Feature - Long Click Listener	2018-06-26 16:30
Fix - Remove Timezone Fix	2018-06-26 13:00
Feature - CalendarMode getter	2018-06-25 19:05
Feature - Sample Dynamic Height	2018-06-25 19:29
Fix - Bug when changing TimeZone of the device (#622)	2018-06-25 17:38
Feature - Sample - Show/Hide Weekdays	2018-06-22 18:26
Chore - Use appcompat versions for customviews (#782)	2018-06-22 18:21
Update MaterialCalendarView.java (#717)	2018-06-22 15:29
Allow set formatted day content description (#779)	2018-06-22 13:22
Skip weekday views on Talkback navigation (#780)	2018-06-22 13:11
Add option to hide week days row.	2018-06-14 06:57
Chore - General Cleanup	2018-05-16 13:38
Update WeekDayFormatter.java	2018-06-04 00:17

HO
Commit 28afa921 (open)
by Halim Omar, 2018-06-14 06:57
committed by Quentin Colle, 2018-06-22 12:56
parent c504fa8c
child 99382b3b

Add option to hide week days row.

Name	Modifi...	Relative Directory
attrs.xml	Modified	library/src/main/res/values
CalendarPagerAdapter.java	Modified	library/src/main/j...terialcalendar
CalendarPagerView.java	Modified	library/src/main/j...terialcalendar
MaterialCalendarView.java	Modified	library/src/main/j...terialcalendar
MonthPagerAdapter.java	Modified	library/src/main/j...terialcalendar
MonthView.java	Modified	library/src/main/j...terialcalendar

Changes of MaterialCalendarView.java (Modified)

```
                newState()
                        .setFirstDayOfWeek(firstDayOfWeek)
                        .setCalendarDisplayMode(CalendarMode.values()[calendarModeIndex])
                        .commit();

                final int tileSize = a.getLayoutDimension(R.styleable.MaterialCalendarView_
 85 lines hidden  show
            if (isInEditMode()) {
                removeView(pager);
                MonthView monthView = new MonthView(this, currentMonth, getFirstDayOfWeek()
                monthView.setSelectionColor(getSelectionColor());
                monthView.setDateTextAppearance(adapter.getDateTextAppearance());
 24 lines hidden  show
            pager.setId(R.id.mcv_pager);
            pager.setOffscreenPageLimit(1);
            addView(pager, new LayoutParams(calendarMode.visibleWeeksCount + DAY_NAMES_ROW)
        }

 563 lines hidden  show
```

```
                        .setFirstDayOfWeek(firstDayOfWeek)
                        .setCalendarDisplayMode(CalendarMode.values()[calendarModeIndex])
                        .setShowWeekDays(showWeekDays)
                        .commit();

                final int tileSize = a.getLayoutDimension(R.styleable.MaterialCalendarView_
 85 lines hidden  show
            if (isInEditMode()) {
                removeView(pager);
                MonthView monthView = new MonthView(this, currentMonth, getFirstDayOfWeek(
                monthView.setSelectionColor(getSelectionColor());
                monthView.setDateTextAppearance(adapter.getDateTextAppearance());
 24 lines hidden  show
            pager.setId(R.id.mcv_pager);
            pager.setOffscreenPageLimit(1);
            int tileHeight = showWeekDays ? calendarMode.visibleWeeksCount + DAY_NAMES_ROW
            addView(pager, new LayoutParams(tileHeight));
        }

```

Figure 2.3: Example of the history of commits visualized wit *SmartGit Tool*

The tool also provides the avatars or images of the commit author (author only) on each commit, and if ⟨gravatar.com⟩[8] access is disabled, colored gravatars will be rendered from the author's initials. The commit date by the author, the committer, the commit date by the committer, the parent commit, and the child commit are also available. In each commit, *SmartGit* also shows the files that have been created or modified and the diffs between them.

[8]Gravatar is a service for providing globally unique avatars and was created by Tom Preston-Werner. Since 2007, it has been owned by Automattic, having integrated it into their WordPress.com blogging platform.

2.6.3 Seart Tool

The Seart tool (DABIC; AGHAJANI; BAVOTA, 2021) serves as a valuable resource for simplifying the sampling of GitHub projects.[9] Designed for empirical studies, Seart GitHub facilitates the search for projects within repositories by applying various project selection criteria. To effectively select projects on GitHub, defining these selection criteria is necessary. This is achieved by using an Application Programming Interface (API) to execute these "queries". However, it is worth noting that the API has usage limitations, such as allowing only 30 requests per minute and providing limited information, e.g., the number of commits in a repository is not included.

The Seart tool is capable of extracting approximately 25 characteristics from GitHub repositories. Widely utilized as selection criteria in project mining studies, this tool proves invaluable for sampling projects' metadata, including details such as the adopted license, number of commits, contributors, issues, and pull requests. It extends support to projects written in ten programming languages, namely Python, Java, C, C#, Objective-C, JavaScript, TypeScript, Swift, and Kotlin (DABIC; AGHAJANI; BAVOTA, 2021). Figure 2.4 illustrates the user interface of the Seart tool.

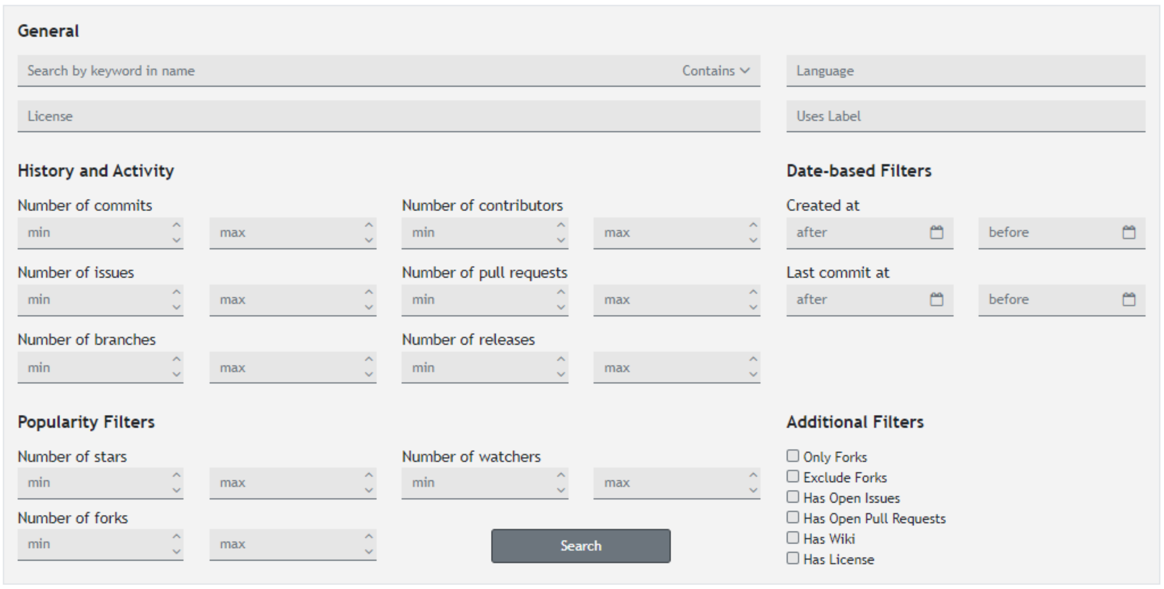


Figure 2.4: Seart Tool overview

[9]⟨https://seart-ghs.si.usi.ch/⟩

Chapter 3

RELATED WORK

This Chapter reviews the existing literature on the key topics we studied in this thesis, highlighting key findings and identifying gaps that this thesis aims to address. Section 3.1 presents some studies addressing the classification of developers' experience. Section 3.2 discusses studies addressing the understanding of developers' perception of test smells. Section 3.3 presents studies exploring the relationship between developers' experience and software refactoring. Section 3.4 discusses studies investigating the survivability of test smells. Finally, Section 3.5 addresses studies discussing data from the StackExchange network about code smells and refactoring.

3.1 CLASSIFYING DEVELOPERS' EXPERIENCE LEVELS

Joblin *et al.* (2017) explored developers' classification based on role count and validated these classifications through a survey involving 166 developers. The authors also introduced a relational perspective on developer roles by analyzing developer networks and investigating developer positions and stability across 10 OSS projects. The findings revealed significant differences between core-peripheral count-based developers and network-based developer classifications. Specifically, network-based classifications demonstrated higher positional stability, elevated positions within the hierarchy, and increased levels of coordination among core developers.

In their study, Crowston *et al.* (2006) aimed to create a clear distinction between core developers, co-developers, and peripheral developers. To achieve this, the authors compared three different approaches for identifying the core group: 1) using a named list of developers, 2) applying Bradford's law analysis, which identifies the most frequent contributors as the core, and 3) utilizing social network analysis to categorize developers into a *core and peripheral* framework. The study focused on analyzing interactions related to bug fixing across 116 Sourceforge projects. The findings revealed that each of the three techniques identified different developers as core members. Furthermore, all three techniques indicated that the core group represented only a fraction of the total number of contributors. The authors emphasized that determining the core developer within a group of co-developers and peripherals is a critical aspect of empirical research for various reasons. One such reason is that while the team structure proposed in the onion model appears valid, it has not yet been thoroughly tested across multiple projects.

In their study, Terceiro, Rios and Chavez (2010) conducted an observational analysis using data obtained from the repository histories of seven projects (aolserver, apache, cherokee, fnord, lighttpd, monkeyd, weborf). The authors gathered information regarding the structural complexity of each change and distinguished between changes made by core and peripheral developers. Their findings indicated that core developers generally exhibited lower structural complexity compared to peripheral ones. This difference was particularly evident in activities aimed at reducing complexity, where core developers tended to excel.

Bird *et al.* (2010) conducted a study exploring the correlation between various measures of ownership and software faults across different domains, including Windows Vista, Eclipse Java IDE, and the Firefox browser. The analysis encompassed two versions of each project, revealing that core developers were associated with both pre-release faults and post-release failures. The study also investigated whether project managers were aware of code contributions from developers with insufficient relevant experience. The authors emphasized that each project had distinct characteristics and employed different development processes, suggesting that the effects of ownership might be linked to the specific development style of each project.

In contrast, our study diverges by focusing on the developer's experience within the project from a test smells perspective. We analyzed the evolution of test classes containing test smells, aiming to uncover whether experienced developers significantly influenced the introduction and removal of test smells throughout the software evolution process.

3.2 UNDERSTANDING DEVELOPERS' PERCEPTION OF TEST SMELLS

Bavota *et al.* (2012) conducted an empirical study on the presence of test smells across 18 software systems, alongside examining software developers' perceptions regarding the impact of test smells on code quality. Next, the authors expanded upon this research in

(BAVOTA *et al.*, 2015), analyzing 27 software systems and surveying 61 developers. The findings revealed a widespread presence of test smells, highlighting potential challenges related to the comprehensibility and maintainability of both test suites and production code.

Likewise, Tufano *et al.* (2016) conducted a survey involving 19 developers to assess their ability to identify instances of test smells within software projects. The findings revealed that developers generally struggle to recognize test smells and seldom take steps to eliminate them from the test code. Additionally, Silva-Junior *et al.* (2020) and Junior *et al.* (2021) carried out empirical studies to uncover the extent to which software developers consciously introduce test smells. The results indicated that even experienced professionals often introduce test smells during their daily programming tasks, although they follow standard practices their organizations ask them to use.

Spadini *et al.* (2020) argued that developers only occasionally view test smells as problematic due to the absence of clear thresholds for interpreting them. To address this, the authors established thresholds for nine specific test smells and empirically evaluated the perception of 31 developers regarding these proposed thresholds. The findings indicated that participants' perceptions aligned with the pre-defined severity thresholds and emphasized the impact of test smells on the maintenance of test suites.

Conversely, Bai *et al.* (2022) explored the impact of test smells on test learning with a group of 42 computer science students. The results revealed that certain test smells either decreased in severity or were less frequent with the evolution of testing frameworks.

Our study differentiates itself by investigating how developer experience affects test quality and the introduction of test smells. In contrast, the others focus more on how developers perceive and deal with test smells and the impact of test smells on test code.

3.3 EXPLORING DEVELOPERS' EXPERIENCE AND REFACTORING

Kim, Zimmermann and Nagappan (2014) discussed the challenges of refactoring within Microsoft, drawing from observations gathered through three empirical studies: a survey, semi-structured interviews involving 328 software engineers, and quantitative analysis of version history data. The authors assessed the impact of refactoring across various software metrics, including defects, module dependencies, code change size and locality, complexity, test coverage, and organizational metrics.

The survey findings led the authors to conclude that the practical definition of refactoring extends beyond a strict interpretation of code transformations that preserve semantics. Developers reported significant costs and risks associated with refactoring, highlighting the need for support beyond automated refactoring tools within Integrated Development Environments (IDEs).

Through interviews, the authors also revealed several years of refactoring efforts focused on Windows. Quantitative analysis of Windows 7 version history demonstrated that the top 5% of refactored modules experienced the most substantial reductions in inter-module dependencies and complexities.

AlOmar *et al.* (2020, 2021) emphasized the significance of incorporating developers' experience into solutions for code refactoring. The authors conducted an empirical study across 800 open-source projects to explore the connection between developers' experience and the frequency of refactoring activities.

Their findings revealed that while numerous developers engage in refactorings, only a small subset is responsible for refactoring both production and test code. Additionally, the authors reported no notable correlation between developers' experience levels and their motivation for refactoring.

Chen, Embury and Vigo (2023) explored whether professionals consider test smells as sources of technical debt and whether investing time and effort in removing them is worthwhile. They assessed the relevance of 19 test smells across 12 open-source software projects and found varying degrees of removal frequency among these smells, with some persisting in the test code for extended periods. This led the authors to conclude that the test smells commonly studied by the research community may not fully capture developers' concerns about test quality. Their study highlights the need for better alignment between academic research and the practical considerations of software development.

In contrast, our study investigates the relationship between human factors (like knowledge and experience) and test code quality, focusing on how test smells can impact software quality. Our aim is to provide guidelines to help developers avoid inserting test smells when creating unit test cases. Both studies contribute significantly to understanding and improving software development practices, although they approach different aspects of the field.

3.4 SURVIVABILITY OF TEST SMELLS

Spadini *et al.* (2020) explored the severity classification of four test smells and investigated how developers perceive their impact on the maintainability of test suites. Similarly, Silva-Junior *et al.* (2020) conducted an empirical study to understand if test professionals unintentionally insert test smells. They surveyed sixty professionals from various companies to analyze the frequency of using a set of 14 test smells widely discussed in the literature and implemented in test smell detection tools.

The results from both studies reveal that experienced professionals introduce test smells during their daily programming activities, even when following their companies' standard practices, rather than solely based on their assumptions. Additionally, the authors found that professional developers' experience cannot be pinpointed as the primary reason for inserting test smells into test code.

Tufano *et al.* (2016) conducted a comprehensive empirical investigation to analyze the occurrence, survivability, and potential association with design issues in production code (code smells) of five test smells within source code. Their study revealed that test smells show high survivability as developers frequently introduce them during test case creation and rarely remove them afterward.

In contrast, our study takes a broader approach to analyzing test code quality. Firstly, we detect a total of 16 test smells. Secondly, we conduct an automated analysis to detect these test smells in each version, followed by manual validation of refactorings to ensure the removal of these test smells. Thirdly, we categorize developers into core and peripheral groups to explore the relationship between their experience in the project and the quality of the test code.

In their empirical study, Kim, Chen and Yang (2021) analyzed 12 real-world open-source projects to explore the evolution and maintenance of test smells, along with their relationship with software quality. Their findings revealed several key points: (i) as systems evolve, the number of instances of test smells increases, while the density of test smells decreases; (ii) approximately 83% of removed test smells were linked to feature maintenance activities, 45% were attributed to refactorings transferring to other test cases, and only 17% were related to specific test smells like Exception Catch/Throw and Sleepy Test; and (iii) test smell metrics were found to correlate with post-release defects, with most test smells having minimal impact on post-release defects. The study also provided insights into developers' perceptions of test smells and current management practices.

Spadini *et al.* (2020) and Silva-Junior *et al.* (2020) focus on developers' perception of test smells and show that even experienced professionals introduce these smells, regardless of their experience. Tufano *et al.* (2016) and Kim, Chen and Yang (2021) explore the evolution and survival of test smells in code, revealing their high persistence and relationship with design problems and post-release defects. Our study aims to analyze whether developers' experiences and skills affect test code quality during software evolution. In addition, our study provides a set of guidelines for avoiding the introduction of test smells.

3.5 DEVELOPERS' PERCEPTIONS THROUGH STACK EXCHANGE TOPICS

Several studies have analyzed discussions within the Stack Exchange network about specific topics in the Software Engineering field (OPENJA; ADAMS; KHOMH, 2020; CHOI *et al.*, 2015; TAHIR *et al.*, 2018; TIAN; LIANG; BABAR, 2019; TAHIR *et al.*, 2020). For instance, Openja, Adams and Khomh (2020) conducted an analysis of 260,023 release engineering questions using topic modeling. Their findings revealed that software testing emerged as the most challenging topic and a crucial contributor to software quality assurance, among the 38 release engineering topics discussed by developers on Stack Overflow.

Other discussions focused on identifying smells in various software artifacts. Choi *et al.* (2015) conducted a preliminary study based on 925 discussions on Stack Overflow concerning code clones. The results highlighted a predominant interest in clone refactoring tools and the need for enhanced support in this area. Similarly, Tian, Liang and Babar (2019) explored developers' perceptions of architectural smells through 207 discussions on Stack Overflow. They found that developers often describe architectural smells using general terms, attributing them to violations of architectural patterns and design principles.

Furthermore, Tahir *et al.* (2018) analyzed 17,126 Stack Overflow posts and manually examined the top 100. Their study revealed that developers frequently seek general assessments of code smells rather than specific refactoring solutions on Stack Overflow. In a related study, Tahir *et al.* (2020) investigated discussions about code smells and anti-patterns across three technical Stack Exchange sites. They observed that developers often discuss the drawbacks of implementing specific design patterns and identify them as potential anti-patterns to avoid.

Regarding code refactoring to enhance overall design quality, Pinto and Kamei (2013) analyzed Stack Overflow posts to study discussions about refactoring tools. They discovered that developers prefer multi-language refactoring tools but face usability issues and a lack of trust in the refactoring process, leading to limited usage. Additionally, Peruma *et al.* (2022) analyzed 9,489 refactoring discussions on Stack Overflow to investigate trends and challenges in refactoring software artifacts. They identified challenges in writing test cases, particularly in adapting them to refactored production code, highlighting key challenges and conclusions for stakeholders across various topics such as code optimization, architecture, unit testing, tools and IDEs, and database.

All of the above authors explore various topics, such as code clones, code smells, architectural smells, refactoring, and design patterns. These works use quantitative and qualitative analysis of posts to identify challenges related to tools and practices in different software artifacts. In contrast, our study focuses on discussions about test smells and test code refactoring, the main challenges developers encounter, strategies, and suggestions on how to refactor test code and deal with post-refactoring behaviors.

Chapter

4

DEVELOPERS' PERCEPTION ON THE SEVERITY OF TEST SMELLS

This chapter presents the first empirical study we carried out in this thesis, aiming to answer **RQ1** and **RQ5** of this thesis. This empirical study has brought the following contributions: (i) an empirical study showing that most developers perceive test smells as low or middle in severity for their software projects, and (ii) initial insights into the consequences of the test smells after refactoring (CAMPOS; ROCHA; MACHADO, 2021).

We structured this chapter as follows: Section 4.1 introduces this empirical study and contextualizes the research problem, the objectives, and a brief description of the methodology used. Section 4.2 details the study methodology. Section 4.3 presents the achieved results. Section 4.4 discusses the main findings. Section 4.5 addresses the threats to validity. Finally, Section 4.6 provides concluding remarks.

4.1 INTRODUCTION

Aligned with the need to understand whether and how harmful test smells are to a software project, many researchers have empirically evaluated the likely effects of test smells (MESZAROS, 2007; GAROUSI; KÜÇÜK, 2018; SILVA-JUNIOR *et al.*, 2020; SPADINI *et al.*, 2020). Garousi and Küçük (2018) enlisted the main negative consequences of the presence of test smell in test code: smells reduce test changeability, stability, readability, and maintainability. Silva-Junior *et al.* (2020) surveyed software testing professionals to understand and analyze how often they insert test smells into the test code and the reasons behind it. The authors showed that, independently of the professional experience, they are prone to insert test smells in their test code, even when using standard practices.

Although some studies claim that test smells might hinder code maintenance, there is still a lack of studies that capture the developers' perception of those issues. In this context, this study investigates how developers recognize test smells. We aim to empirically analyze how developers perceive the severity of test smells in the test code they develop. We refer *severity* to the degree to which a given test smell can negatively impact the test code. In addition to severity, we were also interested in investigating how the test code would behave after removing the test smells.

We conducted an empirical study encompassing three key phases. In the first, we selected six Open-Source Software (OSS) projects from GitHub. The second phase consisted of interviews with the developers of the selected projects to investigate their perception of test smells. During the interviews, we introduced the definitions of each test smell and asked the developers whether they could refactor them. The third step was the analysis of the gathered data.

4.2 RESEARCH METHODOLOGY

We defined two research questions in this study:

RQ1. How do software developers consider the severity of test smells and their effects on test code quality? We aim to analyze how the software developers perceive the impact of the presence of test smells in test code, in terms of quality attributes such as comprehensibility and maintainability.

RQ2. What is the behavior of test code after refactoring test smells? We aim to analyze the changes in the test code after refactoring test smells.

4.2.1 Study Design

The design employed in this study consists of three phases: *Repository Creation*, *Interview*, and *Data Analysis*. Each phase comprises a set of steps, as Figure 4.1 shows.

Phase 1 - Repository Creation. This phase comprised the following steps:

- **Step 1. Filtering by *JNose Test* limitations.** In this step, we used a dataset composed of 21,482 projects available in public GitHub repositories. First, we manually filtered each of the repositories with the following criteria: projects written in Java, composed of at least two stars, with Issues, and test cases created with the JUnit framework (versions 4 or 5). The choice for projects written in Java and

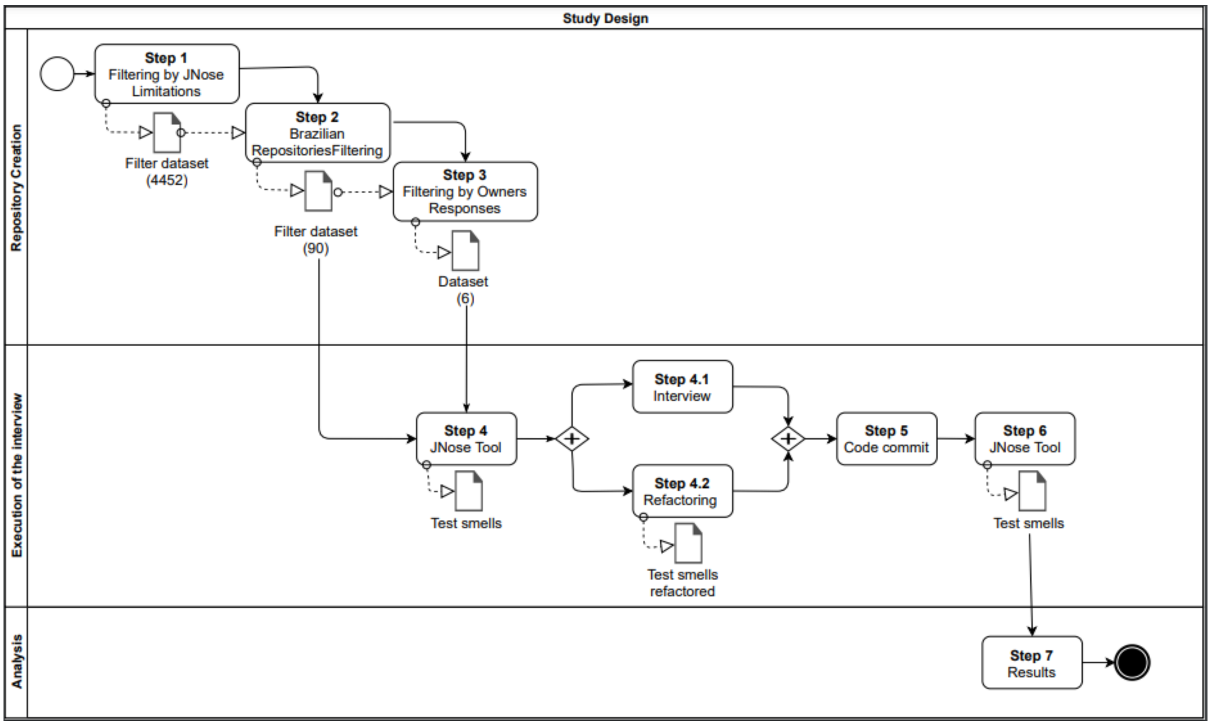


Figure 4.1: Empirical Study Design

tests with JUnit was due to the *JNose Test* limitations. As a result, we obtained a sample with 4,452 public repositories.

- **Step 2. Filtering repositories by ownership.** In this step, we collected the URLs of the 4,452 projects and manually opened each one. First we want to investigate at the Brazil level, we filtered by projects that contained e-mails with Brazilian developers. Then, we ran each of these projects in the *JNose Test* to validate them. As a result, we obtained a sample with 90 public repositories.

- **Step 3. Filtering repositories by responses**. In this phase, we sent out e-mails to the 90 selected projects. We received six positive replies, which composed our dataset.

Phase 2 - Interview. The interview considered each of the developers individually. It comprised the following steps:

- **Step 4. Executing the JNose Test**. The first step was to execute the selected projects in the *JNose Test* (VIRGÍNIO *et al.*, 2020a) to gather evidence about the test smells identified in each project. Based on the yielded results (e.g., the number of test smells present in each project), we created the protocol for the interviews.

- **Step 4.1. Carrying out the interviews**. Next, we proceeded with the actual interviews. First, we carried out the characterization of the participants, through an online questionnaire using Google Forms, which included gathering information on their knowledge about creating and maintaining test cases. The interviews were held virtually, through Google Meet, and guided by the protocol created after detecting test smells in each of the projects. We presented the *JNose Test* to the participants during the interviews, explained that we ran the tool in their project, and presented the test smells we detected in each project. We also showed the most frequent test smells.

- **Step 4.2. Refactoring test code**. After analyzing the presence of test smells in the test code, they had to discuss (i) whether the test smells were harmful to their projects and (ii) whether they should refactor them. In this step, the developers had to refactor at least one test smell. We did not impose which test smell they must refactor. They could select any test smell they wanted.

- **Step 5. Committing test code changes**. After refactoring, we asked the developers to commit the changes to their GitHub repositories. This step was essential to allow us to rerun the *JNose Test* on the refactored project.

- **Step 6. Rerunning the JNose Test**. In this final step, we ran the project again (in the *JNose Test*) to compare the test classes before and after refactorings.

Phase 3 - Data Analysis. This phase comprises the analysis of gathered data, as follows:

- **Step 7. Results**. In this step, we present the results of our empirical study. This research has an exploratory character with a qualitative analysis. For open-ended questions that could combine multiple answers, the sum of percentages could be greater than 100%.

4.3 RESULTS

4.3.1 Developers' Profiles

In this study, we interviewed the developers of each selected project. We next describe their background.

They all hold a Bachelor's degree (two in Computer Engineering, two in Electrical Engineering, and two in Business Information Systems), and two out of them have a

Table 4.1: Selected projects

Project	LoC *	Test Methods	Test Classes	Project URL
Our Digital Bank	84	7	4	⟨https://bit.ly/3ABZui2⟩
csv2bib	98	5	2	⟨https://bit.ly/3xlHyXf⟩
CursoMassa	261	32	4	⟨https://bit.ly/3dQEwSV⟩
Dependency injection	390	23	5	⟨https://bit.ly/36h3ff3⟩
l2jserver	538	31	5	⟨https://bit.ly/3hInK9I⟩
Fatiador	2,362	162	17	⟨https://bit.ly/36gtS3F⟩

(*) Number of LoC in the test classes.

Master's degree. They work in different Brazilian states: three of them work in São Paulo, and the others work in the states of Rio Grande do Sul, Bahia, and Santa Catarina.

Regarding their background, three had over ten years of professional experience, two had less than ten years, and one developer had less than five years. Among the three developers with more than ten years of experience, two of them also had more than ten years of experience in software testing. Among the two developers with up to 10 years of development experience, one also had a strong background in software testing. The remainder have between one and five years of experience in software testing.

4.3.2 Characterization of the selected projects

The six projects we analyzed vary in size - considering the number of Lines of Code (LoC) and the number of test methods, as Table 4.1 shows. The largest project is the `Fatiador` (the project aims to convert flat strings to Java objects and vice versa) with 162 test methods. For the `csv2Bib` (the project aims to convert `.csv` files into `.bib` or `.ris` files) and `Our Digital Bank` (the project provides developers with an API) projects, we could only select two test smell types each.

4.3.3 How do software developers consider the severity of test smells and their effects on test code quality? (RQ1)

We made an attempt to analyze how the developers would perceive the impact of the presence of test smells in test code. To accomplish that, we first considered the degree of severity of each test smell. Table 4.2 summarizes the data about the severity of the test smells identified in each project.

The *AR* test smell was the one that varied the most in terms of severity. Three projects presented variations for this test smell. The project `Curso Massa` (this project aims to

Table 4.2: Degree of Severity of Test Smell per Project

Curso Massa		**l2jserver**		**Dep. Injection**		**Fatiador**		**csv2bib**		**Our Digital Bank**	
TS	Severity	TS	Severity	TS	Severity	TS	Severity	TS	Severity	TS	Severity
AR	Low	AR	Low	AR	Middle	SE	Low	AR	Low	AR	Low
AR	Low	AR	Low	AR	High	SE	Low	AR	Low	AR	Low
AR	Middle	AR	High	AR	Low	SE	Low	RA	Low	EpT	Low
ET	Low	ET	Low	ET	Low	AR	Low	RA	Low	EpT	Low
ET	Low	ET	High	ET	Low	AR	Low	CI	Low	UT	Middle
ET	Low	ET	Low	ET	Low	AR	Low	CI	Low	UT	Low
LT	Low	LT	Low	LT	Low	LT	Low	-	-	-	-
LT	Low	LT	Low	LT	Low	LT	Low	-	-	-	-
LT	Low	LT	Low	LT	Low	LT	Low	-	-	-	-

Legend: Assertion Roulette (AR), Eager Test (ET), Empty Test (EpT), Lazy test (LT), Redundant Assertion (RA), Constructor Initialization (CI), Unknown Test (UT), Sensitive Equality (SE).

support programming classes) presented 2 degrees of severity for the *AR* test smell: *low* and *middle*; the project `l2jserver` (this is a rewrite of the l2jserver project[1]) also presented 2 degrees of severity: *low* and *high*; and the project `Dependency Injection` (this project implements a dependency injection framework in Java with annotations) presented 3 degrees of severity: *low*, *middle*, and *high* severity.

In addition, the *UT* and *ET* test smells also presented differences in their perceived severity, ranging from *low* to *middle* and *low* to *high*, respectively. The remaining projects and test smells only presented a *low* severity degree. Although most of the interviewees classified *AR* test smell as *low* severity, we observed small variations. For example, the developer of the project `Dependency Injection` analyzed 3 test smells of the *AR* type and gave a different classification for each. For this developer, *AR* could impact different her project in different ways.

We asked the developers about the test smells introduced in their respective projects. We wanted to know whether the test smells affect the maintainability of the test code. We next present the main findings.

Two projects (`Fatiador` and `csv2bib`) reported that no test smells found in the projects would affect the maintainability or cause any impact on the test code. Table 4.3 refers to the other four projects, and shows the that *AR* is the test smell with the most negative impact, followed by the *ET* and *UT*.

For the `Curso Massa` project, the developer reported that the *AR* test smell would affect maintainability over time. Conversely, for *ET* and *LT*, the developer reported that they do not affect test code maintainability. Therefore, for the `Curso Massa`, `l2jserver`,

[1]https://bitbucket.org/l2jserver/

`Dependency Injection` projects, the developers reported that the *AR* test smell has a negative impact on the test code. For the `l2jserver` and `Dependency Injection` projects, the developers reported that the *ET* test smell has a negative impact on the test code. Regarding the `Our Digital Bank` project, the developer considered that only *UT* negatively impacts the test code.

Although they have informed us that a few test smells might negatively impact the code, 5 out of 6 developers reported that performing test smell maintenance in the test code would improve the comprehension and evolution of the system.

Table 4.3: Maintainability and impact of test smells per project

CursoMassa			**l2jserver**			**Dep. Injection**			**Our Digital Bank**		
TS	**M**	**I**	**TS**	**M**	**I**	**TS**	**M**	**I**	**TS**	**M**	**I**
AR	Yes	Negative	AR	Yes	Negative	AR	Yes	Negative	AR	No	-
AR	Yes	Negative	AR	Yes	Negative	AR	Yes	Negative	AR	No	-
AR	Yes	Negative	AR	Yes	Negative	AR	Yes	Negative	EpT	No	-
ET	No	-	ET	No	-	ET	No	-	EpT	No	-
ET	No	-	ET	Yes	Negative	ET	Yes	Negative	UT	Yes	Negative
ET	No	-	ET	No	-	ET	No	-	UT	Yes	Negative
LT	No	-	LT	No	-	LT	No	-	-	-	-
LT	No	-	LT	No	-	LT	No	-	-	-	-
LT	No	-	LT	No	-	LT	No	-	-	-	-

Legend: Maintainability (M), Impact (I).

4.3.4 What is the behavior of test code after refactoring test smells? (RQ2)

Before the interviews, we ran the *JNose Test* for each project. The tool generated a `csv` file containing the test smells identified in the projects. During the interviews, we asked the developers to refactor some test smells and next commit the changed test files to their GitHub repositories. Then, we introduced each test smell and asked if they could refactor them. After the modifications, we ran the *JNose Test* for each project again, obtaining a new `csv` file with the latest results. The results are as follows.

Dependency Injection. We selected 9 test smell instances: 3 *AR*, 3 *LT*, and 3 *ET*. From the *AR* test smells, he decided to refactor two out of three into two distinct test classes. The developer successfully removed the test smells from the test code. From the three *LT* test smells, the developer refactored one smell, but she did not remove it. Particularly, for the *LT* test smell, the developer should place the creation of an object in the *setUp()* method, but she linked the *LT* test smell to the use of external components. To refactor, she made use of mocks. Hence, she performed an incorrect refactoring. We also observed that the number of LoC and the number of test methods did not change after refactoring this project. The other test classes, methods, and smells remained without changes.

Curso Massa. We selected 9 test smell instances: 3 *AR*, 3 *ET* and 3 *LT*. The developer decided to refactor the *AR* ones. From the *AR* test smells refactored by the developer, only one was removed. The remaining *AR* were refactored but not removed. To refactor *AR* smells, it is necessary to include an explanation (message) in the first parameter of the *assertion* (e.g., `assertTrue([message,] boolean condition)`). The developer placed the message as the last parameter instead.

Csv2bib. We selected 6 test smell instances: 2 *AR*, 2 *Redundant Assertion (RA)*, and 2 *Constructor Initialization (CI)*. The developer decided to refactor 1 *CI* test smell. After refactoring, he removed the test smell. The *CI* test smell occurs when a test method implements a constructor. Initializing all the fields within the *setUp()* method is necessary. Therefore, for that test smell, a possible refactoring would be to remove the constructor method since it had no instructions. However, the developer attempted to refactor the constructor method by including instructions for setting variables used in other test methods. The *JNose Test* continued to consider it a *CI* test smell. Since the developer modified the test class, the LoC and the number of test methods changed. For the `Run.java` test class, its LoC increased from 44 to 57, and the number of test methods risen from 2 to 5.

Fatiador. We selected 9 test smell instances: 3 *SE*, 3 *AR*, and 3 *LT*. The developer decided to refactor 2 *AR* smells from 2 different test classes: `DecimalWriter.java` and `IntegerWriter.java`. After refactoring, the developer successfully removed the two smells. For each test class, he added 1 LoC each (from 47 to 48 LoC and from 40 to 41 LoC, respectively. The other test smells did not present any change.

l2jserver2. We selected 9 test smell instances: 3 *AR*, 3 *ET*, and 3 *LT*. The developer refactored 2 *AR* and 1 *ET*. In addition, he also refactored other *AR* test smells not previously selected. For example, the `BitSetIDAllocator.java` test class contained 7 *AR* before refactoring. Although we asked him to refactor just 1 of them, he refactored 3 occurrences of this test smell. In total, he refactored 4 *AR* test smells for 2 different classes (`BitSetIDAllocator.java` and `CharacterIDProvider.java`), in which all of them were successfully removed after refactoring.

For the *ET* test smell, the developer chose to refactor the smell inside the CharacterIDProvider.java test class. Therefore, it was necessary to remove the multiple calls to the multiple production methods. However, the developer did not perform the necessary removals. Moreover, during the incorrect refactoring of the *ET* test smell, he introduced a new type of smell in the project, the *LT*, which occurs when multiple test methods call the same production method. Thus, 2 test classes (`CharacterIDProvider.java` and `WorldServiceImpl.java`) have changed. The first increased in 1 LoC, while the latter decreased by 1. For the number of test methods and other test smells, there was no change in behavior.

Our Digital Bank. We selected 6 test smell instances: 2 *AR*, 2 *Empty Test (EpT)*, and 2 *Unknown Test (UT)*. The developer decided to refactor 1 *AR*, 2 *EpT*, and 2 *UT* test smells. This project contained 4 test classes. During the refactoring, the developer removed 2 of them. In addition, the `Validator.java` test class contained 35 LoC, and 90 LoC after refactoring. Also, the number of test methods increased from 4 to 8. All the 3 types of test smells, *AR*, *UT*, and *EpT* presented different behaviors after refactoring: the amount of *AR* test smells increased from 2 to 9; the 2 *UT* refactored were successfully removed, and 2 *EpT* smells were refactored and removed. The project had 2 *EpT*, 1 was removed after refactoring and the other after removing one of the test classes. Additionally, after refactoring, the developer added other 9 test smells, 7 *Ignored Test (IT)* and 2 *LT*. From the analyzed projects, `Our Digital Bank` was the one that had the most changes in the test code, which had decreased the number of test classes, increased the number of test smells, and the number of LoC and test methods. After refactoring, two test smells that did not exist before in the project were added to the test code.

4.4 DISCUSSION

In this study, we considered three degrees of severity (low, middle, and high) for the test smells. We used this approach to evaluate eight test smells (*AR*, *EpT*, *UT*, *ET*, *LT*, *CI*, *Sensitive Equality (SE)*, and *RA*). Five test smells were considered as *low* severity for all projects: *LT*, *SE*, *EpT*, *RA*, and *CI*. From the set of test smells the interviewees analyzed (48 test smells identified in their projects), 42 were considered as *low* severity (≈87,5%). Only 6 out 48 test smells varied in degrees of severity (≈12,5%), 4 *AR*, 1 *ET*, and 1 *UT*, spread over 4 projects.

The *AR* was the one that varied the most. It was perceived by the developers in the three severity degrees, while the *ET* and *UT* presented two different degrees. According to the developers, the *AR* test smell might harm test code maintainability. For example, one interviewee claimed that *"if someone else is going to test the system, they will not know why the test failed"*. Another interviewee also commented about the *UT* test smells, as follows: *"UT harms my code because it shows that it has unnecessary code, and triggers a function in my system that is not validated as it should"*.

Although most test smells received a low severity degree, the developers reported that, in general, test smells might be indicative of problems and harmful to the test code. On the other hand, they claimed that their systems were relatively small, containing simple test cases, and therefore, the test smells may not have as much impact. For larger projects, on the other hand, they may affect more. Therefore, there is a need to conduct further studies to know if those and other types of test smells have a degree of severity that would be different.

According to the developers, fixing test smells may positively impact the test code by improving its overall quality, making it more comprehensible, and easing the system's evolution. Two out of six developers also believe that refactoring test smells can result in detecting more bugs. Although four developers claim that refactored test smells do not contribute to the detection of bugs.

Moreover, for one developer, the *LT* should not be considered a test smell at all. They argue that it might be necessary to have more than 1 test method testing the same production method. In this case, each test method has a different purpose concerning the production class method. For example, for a given test method, the focus may be to exclude something, while in another, it may be to change something else; and the developer cannot see how to perform the test methods in a single method.

4.5 THREATS TO VALIDITY

Internal validity. We selected projects from 84 to 2,362 lines of code. This subset of projects may not represent industrial software systems, and therefore replications in this context are desirable. Although Deursen *et al.* (2001) addressed 21 test smells, we only selected a subset of 8 test smells in this study. Nevertheless, we selected the most frequent test smells in the projects, gathered from the *JNose Test* results. Furthermore, the choice of Brazilian developers can be a threat to validity, to mitigate this we perform a characterization of the developers' profile and skills in the knowledge of the JUnit Test Framework.

External validity. The number of participants we found willing to participate in the interview represents a threat to the results. Nonetheless, we sent emails to 90 GitHub developers, and six replied. This sample is not representative, and we expect to replicate this study with a more significant number of participants. Although the results cannot be generalized, they represent a partial view of developers' practices. We believe that this preliminary study is significant and indicative of a trend in the area. In addition, we intend to investigate this topic further with a considerable number of test smells and projects. We also present the step-by-step methodology of this work that may allow further replications of this study. Although most of the smells investigated in this study show low severity, we cannot generalize to other projects and other test smells.

Construct validity. The interpretation might influence the research results. Regarding refactoring, we did not provide information on how one developer should refactor test smells. We only provided them with definitions and how to detect each of them. The number of refactored test smells varied in each project because we asked the developers if they could refactor and then which one(s) they would like to refactor. The number of smells refactored depended on the developer's choice. As each project presented different test smells, it was impossible to compare the same test smell in all projects. Since the projects have not been in development for more than two years, this can threaten validity. However, after refactoring, all projects were committed to GitHub without any problems. Another threat to construct validity is the collection of emails from developers via GitHub. This can raise ethical issues, such as the possibility of emails being perceived as spam, resulting in low adherence and compromising the sample's representativeness. To mitigate these risks, it is advisable to adopt a more transparent approach, asking for prior consent and using alternatives such as internal messages on GitHub or issues to engage developers in a less invasive way.

4.6 CONCLUSION

This Chapter presented a study we carried out to analyze how developers perceive the severity of test smells on test code quality. To accomplish this goal, we selected a set of OSS projects from GitHub and interviewed their developers. We also asked them to refactor their test code to remove test smells. Hence, we could observe, from their standpoint, how the developers could consider the effects of such removal.

The results indicated that test smell severity is highly dependent on the context. For example, for the same project, a given test smell may present different degrees of severity. Regarding test smell refactoring, our initial results pointed out that testers may not know how to refactor a test smell to remove it. In addition, we also identified that it is likely that the removal process could induce the inclusion of new test smells.

Chapter

5

UNDERSTANDING PRACTITIONERS' STRATEGIES TO HANDLE TEST SMELLS

This Chapter presents the second empirical study we carried out to answer the **RQ2** of this thesis. In summary, this study has brought a qualitative study on practitioners' experience with the following contributions: (i) strategies for creating and maintaining code and testing; (ii) effects of test smell on the maintenance and test code quality; (iii) test code refactoring strategies different from the strategies currently presented in the literature; and (iv) perception of the severity level of test smells, listing the most harmful test smells to the test code (SANTANA *et al.*, 2021).

We structured this chapter as follows: Section 5.1 introduces this empirical study. Section 5.2 presents the methodology we employed. Section 5.3 presents the results of the survey study. Section 5.4 presents the results of the interview study. Section 5.5 discusses the results of both survey and interview studies. Finally, Section 5.7 concludes the Chapter.

5.1 INTRODUCTION

Although several studies have been investigating test smells (GAROUSI; KÜÇÜK, 2018; VIRGÍNIO *et al.*, 2019; SILVA-JUNIOR *et al.*, 2020; SANTANA *et al.*, 2020; VIRGÍNIO *et al.*, 2020a, 2020b; SPADINI *et al.*, 2020), there is still a lack of industrial studies to understand better how practitioners deal with them in practice. For example, we have found no evidence about studies exploring refactoring strategies suggested by practitioners and how they handle test smells regarding software maintenance and evolution.

Hence, we conducted a multi-method study with software testing practitioners in two phases: survey and interviews. We aim to investigate test smell perceptions and strategies on test code adopted by practitioners, such as (i) test code creation and maintenance, (ii) severity, (iii) refactoring practices, and (iv) techniques suitable to improve the test code.

We applied a survey to 87 software testing practitioners with experience in the JUnit framework. We also performed eight interviews to complement the study and better understand how test smells can affect and refactor the test code. As a result, most participants know test smells and reported two to five different refactoring strategies for each test smell type. Furthermore, they considered three types as the most severe test smells: *Conditional Test Logic (CTL)*, *Unknown Test (UT)*, and *Test Run War (TRW)*.

5.2 RESEARCH METHODOLOGY

This study investigates the effects of test smells on the maintenance and test code quality from the perspective of practitioners with experience in software testing. Therefore, we defined the following research question: ***"What is the practitioners' perception about the test smells effects on maintainability and test code quality?"*** We split this question into five sub-questions, as follows:

RQ1. How do practitioners create and maintain test code? This question aims to understand how developers create and maintain test cases, e.g., manually or through test generator tools (e.g., Randoop[1], EvoSuite[2]). This RQ also investigates test maintenance frequency.

RQ2. Are the practitioners aware of test smells? This question aims to understand the practitioners' knowledge of test smells and their views to analyze whether they affect the test quality and understanding. We also want to investigate whether practitioners know how often they insert test smells into the test code.

RQ3. What is the test smell severity from the practitioners' perspective? This question investigates how much developers estimate the severity of each test smell, i.e., which test smells mainly harm test code.

RQ4. How do practitioners deal with test smells? This question seeks to investigate measures taken by developers when they find test smells in the

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

existing test code. We aim to understand the participants' opinions on the need to refactor test smells and the frequency with which they do so in their projects.

RQ5. What is the practitioners' perception of test quality? This question investigates the participants' views on the importance of improving test code quality, and the main strategies developers could employ to improve the test suite's quality in the software development process.

In this study, we employed a multi-method research approach. Multi-method research combines qualitative and quantitative methods to gather information (YIN, 2017). We applied surveys and interviews so that we could answer our research questions. In the former, we used objective and quantitative questions. In the latter, we designed qualitative questions. The RQ_5 were answered by one method, and the RQ_1, RQ_2, RQ_3, and RQ_4 by both methods. Table 5.1 presents the RQs and topics we investigate with each method. In the following sections, we describe the survey and interview planning.

Table 5.1: Key topics of research questions addressed in the survey and interview.

RQ	**Survey**	**Interview**
RQ_1	Test creation strategies; Test maintenance strategies; Maintenance frequency	When tests are maintained; Reuse of test code; Tools used to generate the test code
RQ_2	Awareness of test smells definition and types; Test smells effects on test code quality/maintenance; Test smells frequency	Test smells effects on test code quality/maintenance; How test smells affect test code
RQ_3	Test smells severity	Classification of the 3 most severe test smells
RQ_4	Opinion on the need to refactor; Frequency of refactorings	Opinion on the need to refactor; Refactorings strategies
RQ_5	—	Importance of evaluating test codes quality; Strategies to improve test quality

5.3 SURVEY

5.3.1 Survey methodology

Identify and characterize target audience. After identifying the research objectives, we recognized the target audience, composed of professionals with experience in testing with the JUnit framework.

Participants sample. We selected only practitioners from Brazil. We selected the sample based on three aspects: convenience, social networking platforms (e.g., LinkedIn), and snowball sampling (WOHLIN, 2014), i.e., the potential participants were also encouraged to send invitations to other people from their network.

Survey design. The survey comprised 35 closed-ended questions. Among these, 16 followed a text field where they could justify their answers, if necessary. We grouped the questions into five sections: (a) conditional filter to our target audience; (b) demographic information; (c) test creation and maintenance; (d) perspective on test practice patterns (test smells); and (e) knowledge about test smells. We designed the survey in Google Forms.[3] The sections were structured as follows:

(a) it had one closed-ended question to select participants with JUnit framework experience;

(b) it had four closed-ended questions corresponding to participants' profiles: where they live, their education level, the amount of time they work in the software industry, and length of experience with the JUnit framework;

(c) it had four closed-ended questions asking participants how tests are created and maintained, test maintenance frequency, and which test code generation tools they use;

(d) for each of the 8 test smells investigated, we present a code snippet and a brief description of the practice without mentioning the test smell name;

(e) we asked two closed-ended questions: whether the participants knew the term "test smell" and which ones they knew.

Pilot survey. We conducted a pilot study with 5 participants: 1 graduate student in Computer Science, 2 Master's students in Computer Science, and 2 Ph.D. students in Computer Science. The goal was to gather feedback on the survey (questions and answer choices completeness) and estimate the average time to complete it. The pilot answers were discarded from the final evaluation.

Distribute questionnaire. We invited potential participants through e-mail and social media like LinkedIn, WhatsApp, Telegram, and Facebook. We distributed the survey from December 3, 2020, to February 25, 2021.

Analyze results and report the results. Three of the authors independently extracted the general themes from all responses to each question. Using these themes, the authors held discussion sessions to develop an agreement to analyze each question. We could classify the answers according to the final coding obtained in the discussion.

[3]⟨https://docs.google.com/forms/⟩

5.3.2 Survey results

After analyzing and removing invalid responses from the survey, we obtained 87 valid responses. Those responses came from 16 different states in Brazil. The most representative states were São Paulo (27.59%) and Bahia (19.54%). The participants had different levels of education: 2 (2.30%) had High school completed, 19 (21.84%) participants were B.Sc. students, 30 (34.48%) were B.Sc., 18 (20.69%) had MBA, 5 (5.75%) were M.Sc. students, 7 (8.05%) M.Sc., and 6 (6.90%) were Ph.D. students.

As a requirement for filling out the survey, we selected participants with experience in software development using the JUnit framework. Therefore, all participants had experience in the industry, and more than half had more than five years of experience. Regarding the JUnit framework, 19.54% had up to 1 year of experience, 56.32% had up to 5 years, 12.64% had up to 10 years, and 11.49% had more than ten years of experience. Nevertheless, we do not perform any comparison between the developers' experiences.

5.3.2.1 Test code creation and maintenance (RQ1)

In this question, we investigate how developers create and maintain tests. We examined whether testers create and maintain tests manually or automatically and the support tools they commonly use.

The possible answer choices for creating and maintaining the test code were *Manual*, *Automated*, and *Manual & Automated* simultaneously. We also included an open field named *Other*. Therefore, we observed that 60.91% of the participants manually create test cases, 32.18% use manual & automated strategies, and 5.74% adopt exclusively automated strategies. For maintaining the test code, 72.41% perform manual maintenance; 21.83% use both manual & automated, and 1.14% only use automated strategies. We also obtained inconclusive answers in the *Others* field (1.14% for creation and 4.59% for maintenance).

Regarding the tools used by testers, they could mention more than one. Of the 87 participants, 78 (89.65%) reported not using any test code generation tools. The remainder reported: 3 participants use EvoSuite, 2 participants use Randoop, 1 participant use Diffblack,[4] and 2 participants use auxiliary libraries. Three participants answered that they used JUnit and Selenium as test code-generator tools. However, these tools do not automatically generate test code and depend on human intervention.

[4]⟨https://www.diffblack.com⟩

Additionally, we investigated how often tests are maintained. We used the 5-point Likert scale: Never (≅ 0%), Rarely (≅ 25%), Sometimes (≅ 50%), Often (≅ 75%), and All the time (≅ 100%). 14.9% of the participants reported they perform maintenance "All the time"; 26.4% informed "Often"; 37.9% "Sometimes"; 20.7% reported they perform maintenance "Rarely". All participants claimed they perform maintenance on test code at least rarely.

5.3.2.2 Awareness about test smells (RQ2)

We asked the participants whether they had already heard about test smells, and we observed that 69% of participants had already heard about them. We also asked about the test smells they were aware of. We present to them a list of 23 types of test smells. 31% of the participants reported not knowing any test smell. However, some have heard of the term although they are unaware of any test smells.

Table 5.2 shows the test smells with their respective percentages (relative and absolute) concerning the participants' knowledge in descending order. In this case, the relative rate refers to the 60 participants who have heard about test smells. The absolute rate refers to the knowledge of all 87 participants. In addition to the 23 test smells on the list, one of the participants mentioned a type of test smell that was not on the list: *Indeterminate Test*.

We also investigated developers' perceptions of test smells' effects on the test code's quality and maintenance. We presented code snippets with test smell and asked if that practice could affect the quality and maintenance of the test code. Participants could answer "Yes," "No," "I don't know," or write a different answer. When the answer was different from *yes*, *no*, and *I don't know*, and it was not possible to classify them in one of these categories, we defined them as *inconclusive*. Table 5.3 shows the opinion of the 87 participants, in which the test smells that most affect code quality were: UT, CTL, AR, and TRW.

We presented a test code snippet for each of the eight test smells introduced in Section 5.3.1. We asked how often the participants write test code that is similar to those test smells. They could choose one of the following answers: Never (≅ 0%), Rarely (≅ 25%), Sometimes (≅ 50%), Often (≅ 75%), and All the time (≅ 100%).

Table 5.4 presents the answers. For instance, 79.31% of the participants responded that they *never* (≅ 0%) write test code similar to the UT test smells, 31.03% of participants responded that they *rarely* (≅ 25%) write test code similar to CTL, 24.14%

Table 5.2: The most popular test smells

Test Smell	**% Relative**	**% Absolute**
Duplicate Assert	71.67%	49.42%
Empty Test	58.33%	40.23%
Redundant Assertion	58.33%	40.23%
Ignored Test	56.67%	39.08%
Exception Handling	55%	37.93%
Conditional Test Logic	53.33%	36.78%
Test Code Duplication	50%	34.48%
Sleepy Test	45%	31.03%
Magic Number Test	43.33%	29.88%
Lazy Test	35%	24.13%
Assertion Roulette	30%	20.69%
Constructor Initialization	28.33%	19.54%
Redundant Print	23.33%	16.03%
Eager Test	21.67%	14.94%
For Testers Only	20%	13.79%
General Fixture	18.33%	12.64%
Default Test	16.67%	11.49%
Resource Optimism	15%	10.34%
Sensitive Equality	13.33%	9.19%
Indirect Testing	13.33%	9.19%
Unknown Test	13.33%	9.19%
Mystery Guest	10%	6.89%
Test Run War	5%	3.44%
Indeterminate Test	1.67%	1.14%

Table 5.3: Participants' opinion whether test smells affect the test

Response	**AR**	**CTL**	**DA**	**EH**	**RO**	**ST**	**UT**	**TRW**
Affects (%)	78.16	80.46	65.52	45.98	62.07	73.56	87.36	78.16
Don't affect (%)	16.09	6.90	19.54	42.53	21.84	17.24	8.05	11.49
Don't know (%)	0.00	0.00	0.00	0.00	1.15	1.15	0.00	1.15
Inconclusive (%)	5.75	12.64	14.94	11.49	14.94	8.05	4.60	9.20

participants responded that they *sometimes* ($\cong$ 50%) write test code similar to EH, 14.94% participants responded that they Often ($\cong$ 75%) write test code similar to AR and DA, and only 13.79% participants responded that they *all the time* write test code similar to AR ($\cong$ 100%).

Table 5.4: Frequency that test smells are inserted

Frequency	AR	CTL	DA	EH	RO	ST	UT	TRW
Never (%)	29.89	56.32	34.48	44.83	34.48	54.02	79.31	64.37
Rarely (%)	27.59	31.03	26.44	19.54	29.89	25.29	12.64	14.94
Sometimes (%)	13.79	11.49	20.69	24.14	18.39	9.20	6.90	10.34
Often (%)	14.94	1.15	14.94	9.20	13.79	11.49	1.15	8.05
All the time (%)	13.79	0.00	3.45	2.30	3.45	0.00	0.00	2.30

5.3.2.3 Severity of test smells (RQ$_3$)

For this research question, participants should also consider code snippets with test smells. We used the 5-point Likert scale to classify severity as "None," "Few severe," "Moderately severe," "Severe," and "Very severe."

Table 5.5 shows the severity level for each test smell. For instance, we observed that 44.83% of participants considered the test smell AR as few severe for the test code; the CTL was considered as very severe by 37.93% of the participants; 34.48% considered the DA test smells few severe, and EH was considered as few severe for 41.38% of the participants.

Table 5.5: Level of severity of test smells

Severity	AR	CTL	DA	EH	RO	ST	UT	TRW
None (%)	6.90	2.30	17.24	14.94	9.20	11.49	13.79	10.34
Few severe (%)	44.83	9.20	34.48	41.38	36.78	19.54	20.69	22.99
Moderately (%)	25.29	25.29	32.18	31.03	20.69	31.03	13.79	12.64
Severe (%)	10.34	25.29	9.20	5.75	22.99	14.94	16.09	22.99
Very severe (%)	12.64	37.93	6.90	6.90	10.34	22.99	35.63	31.03

5.3.2.4 How test smells are handled (RQ$_4$)

We investigated whether developers refactor test smells and how often they do it. We asked if they considered it necessary to refactor the code in the example presented. Table 5.6 shows the results of the participants' opinions about each type of test smells. For instance, for RO, 65.52% of participants stated that they consider it necessary to refactor it, 25.29% said that they did not think it necessary to refactor it, 8.08% gave inconclusive answers, and 1.15% said that they do not know how to answer it.

Table 5.6: Opinion on the need to refactor test smells

Response	AR	CTL	DA	EH	RO	ST	UT	TRW
Refactor (%)	72.41	87.36	63.22	62.07	65.52	77.01	89.66	77.01
Don't refactor (%)	19.54	9.20	27.59	32.18	25.29	17.24	5.75	14.94
Don't know (%)	0.00	0.00	0.00	0.00	1.15	0.00	1.15	2.30
Inconclusive (%)	8.05	3.45	9.20	5.75	8.08	5.75	3.45	5.75

As Table 5.6 shows, there was generally more agreement than disagreement for refactorings. We observed unanimity regarding the refactoring of UT and CTL. 89.66% and 87.36% of participants, respectively, agreed with their refactorings. Meanwhile, the refactorings that most presented disagreements were EH (32.18% of participants) and DA (27.59% of participants) test smells.

Table 5.7 shows the frequency in which each test smell is refactored according to the 5-point Likert scale: Never ($\cong$ 0%), Rarely ($\cong$ 25%), Sometimes ($\cong$ 50%), Often ($\cong$ 75%), and All the time ($\cong$ 100%). 33.33% of participants reported that they *never* ($\cong$ 0%) refactor UT and TRW test smells. On the other hand, for 21.84%, the UT is refactored *all the time* ($\cong$ 100%). We found that the most refactored ones are AR and CTL test smells.

Table 5.7: Frequency of test smells refactoring

Refactoring	AR	CTL	DA	EH	RO	ST	UT	TRW
Never (%)	11.49	20.69	25.29	24.14	24.14	24.14	33.33	33.33
Rarely (%)	24.14	21.84	28.74	42.53	36.78	28.74	29.89	21.18
Sometimes (%)	31.03	25.29	29.89	17.24	18.39	20.69	6.90	17.24
Often (%)	24.14	17.24	12.64	8.05	12.64	13.79	8.05	4.60
All the time (%)	9.20	14.94	3.45	8.05	8.05	12.64	21.84	12.64

5.4 INTERVIEW

5.4.1 Interview methodology

Pilot interview. Before leading the interviews, we also performed a pilot interview to validate our questions and assess the timing, and then this participant's answer was discarded. We interviewed one researcher with a degree in Computer Science and experience in software development and the JUnit framework for this pilot. The pilot interview lasted 19 minutes. Therefore, we noticed the need also to present code snippets with test smells to the interview participants.

Invitation. In the survey, we asked the participants if they could participate in the second stage of this research, and if so, the participants would leave their e-mails. After obtaining survey responses, we sent 31 emails scheduling interviews with participants interested in collaborating with the second phase of the study, while only 8 returned the email confirming the interviews.

Conducting interviews. All interviews were conducted via Google Meet,[5] and we recorded them with the OBS Studio tool[6] for further analysis. During interviews, participants consented to record and agreed to disclose the results under anonymity. Interviews ranged from 40 to 60 minutes. More than one researcher participated in the interviews, but only one conducted the inquiry. As the survey had a more objective and quantitative end, in the semi-structured interviews, we could direct our questions to obtain more detailed information about their thoughts on the topic.

Transcription. After conducting interviews, we listened to the recordings and transcribed the entire conversation. To transcribe the audio, we had the support of the TranscriberBot for Telegram.[7]

Coding Process. We counted on the support of two other colleagues from our research group to conduct the data analysis. We investigated the participants' responses for each question and created the codings to classify and group responses. To create the codes, each person independently extracted the general themes from all questions' responses. Through discussion sessions, we could discuss each question to reach an agreement. Each researcher read, analyzed, and grouped responses individually. We then presented our codings for consensus. In case of disagreements among codings, we analyzed and decided which coding was best suited to those answers.

5.4.2 Interview results

We interviewed a subgroup of survey participants. In the survey, participants expressed interest in collaborating with the second phase of the study; we sent 31 invitations to schedule the interviews, and eight professionals responded, confirming their participation in the interview.

The interviewed participants came from four Brazilian states: São Paulo, Santa Catarina, Pernambuco, and Minas Gerais; six participants had a Bachelor's degree, one had an MBA in technology, and one was a Master's student. Most of them had experience

[5]⟨https://meet.google.com⟩
[6]⟨https://obsproject.com⟩
[7]https://github.com/charslab/TranscriberBot

with development for more than five years. Regarding their experience with JUnit, seven had up to five years of experience, and one participant had up to one year of experience. Table 5.8 shows their characterization. The interviews lasted 43.4 minutes on average, with a standard deviation of 9.3 minutes.

Table 5.8: Profile of interview participants

ID	**Education**	**Industry (years)**	**JUnt (years)**	**Interview**
P01	B.Sc.	Up to 10	Up to 5	$\cong$ 37 min.
P02	B.Sc.	More than 10	Up to 1	$\cong$ 46 min.
P03	M.Sc. student	Up to 10	Up to 5	$\cong$ 53 min.
P04	B.Sc.	Up to 5	Up to 5	$\cong$ 45 min.
P05	B.Sc.	Up to 5	Up to 5	$\cong$ 25 min.
P06	B.Sc.	Up to 10	Up to 5	$\cong$ 48 min.
P07	MBA	More than 10	Up to 5	$\cong$ 53 min.
P08	B.Sc.	More than 10 years	Up to 5	$\cong$ 40 min.

5.4.2.1 Test code creation and maintenance ($\mathbf{RQ_1}$)

In the survey, we asked the participants about the strategies to create and maintain the test code, the frequencies at the tests are maintained, and the test code generation tools they commonly used. For the interview, we asked them when the tests are maintained, which was when they decided to evolve the test and reuse it across the releases.

All participants reported that the tests usually receive maintenance when a new feature is implemented in the production code. In addition, some of them mentioned that tests are maintained when a given modification impacts an existing functionality causing bugs in the system (mentioned by 2 participants); when the test code is inspected for general refactorings (mentioned by 2 participants); when the tests have non-deterministic behavior (mentioned by 1 participant); during continuous deliveries (mentioned by 1 participant). The test code is only removed when a feature is removed from the production code since the test code snippet does not make sense to be maintained anymore.

5.4.2.2 Awareness about test smells ($\mathbf{RQ_2}$)

In the interview, we were also interested in understanding how the test is affected by each type of practice. We asked participants if each test smell presented could affect the code. When the participant answered affirmative, we wondered how that test smell could affect the test code. We next discuss the participants' perception of each test smell we analyzed.

- **AR.** Only one participant reported that this type of practice does not affect the test code. The others said that having multiple assertions without explanations within a method affects the test code's maintenance and quality. It makes reading, understanding the expected result, and reusing the code hard. They informed us that it is preferable to have smaller methods with fewer assertions than a single method with several ones.

- **CLT.** Similarly to the AR, only one participant reported that this type of practice does not affect the test code. According to the others, using conditional structures in the test code affects readability, increases the method's complexity, and makes it difficult to understand the method objective. The method's behavior can be changed using these structures. The test can pass without actually validating correctly, generating tests with non-deterministic behavior. Participants also reported that the single responsibility principle is not followed when using this practice (AMPATZOGLOU *et al.*, 2019).

- **DA.** All participants agreed that it affects the quality/maintenance of the test code. They explained that having assertions with equal parameters in the same method affects the test objective understanding and readability and hurts the single responsibility principle.

- **EH.** Only one participant said that it does not affect at all. The others believe that using *try/catch* in the test method's body makes it difficult to understand the test's purpose, affecting test readability. In addition, a test with *try/catch* structures becomes more verbose, increasing complexity. Participants also reported that this practice could have an impact similar to using conditionals in a test.

- **RO.** All participants agreed that it affects the quality/maintenance of the test. They claimed that using external resources without first checking their status could harm the overall quality. It can be challenging to predict resources' states, as it can present a non-deterministic behavior (Flaky Test (ZOLFAGHARI *et al.*, 2020)), making it hard to identify the root problem. Moreover, it could cause an exception, e.g., a *Null Pointer Exception*.

- **ST.** All participants agreed that it affects the quality/maintenance of the test code. They believe it affects the test because it impacts execution time and performance.

The more `Thread.Sleep` in the test suite, the longer it will take to run it. In some scenarios, waiting can generate non-determinism.

- **UT.** All participants agreed that it affects the quality/maintenance of the test code. According to them, a test method with no assertions does not check anything; the objective of the method is implicit. Therefore, it does not make sense to keep this method. This practice was considered a developer error and violated the principle of validating an automated test.

- **TRW.** All participants agreed that it affects the test code and is not a good practice. Using this type of practice creates a dependency on the order in which tests are performed, violating first principles and allowing cascading error to occur; affects maintenance during the evolution of the system; increases the coupling among tests; creates resource competition among test methods; and impacts the maintenance of the test and production code. It can also make it difficult to identify why the test failed.

5.4.2.3 Severity of test smells (RQ$_3$)

We asked participants to choose a severity category that best suited each test smell type for the survey. In the interview, we asked them again to classify test smells according to severity. We asked participants to report the three worst testing practices (practices should be ordered by severity degree). Only one participant in the interview could not classify, and one reported only the two worst practices.

We counted the number of times the answers mentioned each type of practice (test smell). When there was a tie between two test smells, we used the sort order criterion to break it. For instance, they mentioned CTL and UT four times. Two participants mentioned CTL as the worst practice.

The RO, AR, and DA test smells were mentioned only once. As RO was the second-worst practice and the others as the third-worst practice, we classified RO as the 5th most severe test smell. Meanwhile, we ranked the test smell AR and DA with the sixth position; there was a tie. None of the participants mentioned the practice refers to the test smell EH. They believe TRW, CLT, UT, ST, RO, AR, DA, and EH are the final list of the worst test smells.

5.4.2.4 How test smells are handled (RQ$_4$)

We asked the participants if they considered it necessary to refactor the test code with each practice presented. When the participants agreed to refactor, we asked them which strategy they would use. As refactoring a test smell might depend on the context, the participants could suggest more than one refactoring strategy. We next discuss the strategy for each test smell:

- **AR.** Seven participants agreed they should refactor this test smell, but only six participants suggested refactoring to this test smell. Five participants suggested splitting the method into others to avoid multiple assertions within the same test method, and three suggested including the explanation parameter.

- **CLT.** Seven participants agreed they would refactor CLT, and one reported it as unnecessary to refactor the test code with conditional structures. Four participants reported that they could do the refactoring by splitting the method into more methods to reach the conditional structures. Two participants recommended refactoring with an abstraction of the content of the conditional structure in an auxiliary method. One participant was unable to suggest refactorings.

- **DA.** All participants agreed they should refactor the DA test smell. Five participants suggested refactoring it by dividing the original method into more test methods for each new value that the variable assumes. Three participants were unable to refactor it.

- **EH.** Only one participant reported refilling a test code with try/catch structures was unnecessary. Three participants suggested dividing the test method into more methods, where each method individually validates success and exception. Two participants reported that it was necessary to analyze the production method to make any refactoring decision. One participant would remove the *try/catch* and evaluate the need to insert *throws*. One participant informed that the refactoring could be done with *Assumptions*[8] to ignore the test method under certain conditions. Furthermore, other participants recommended using *assertThrows()* and *assertException()* methods to validate the expected behavior.

- **RO.** All participants agreed they should refactor the practice of using external resources without checking. Six participants informed us that they should do a check

[8]https://junit.org/junit5/docs/5.0.0/api/org/junit/jupiter/api/Assumptions.html

before using the resource. Four participants suggested using abstractions for the resource (e.g., mock). Three participants suggested creating the resource using the *setup* method. One participant claimed he would use *JUnit* resources to handle temporary files. Furthermore, one participant suggested splitting it into more tests to verify a possible exception launch.

- **ST.** All participants agreed they should refactor test code with a `thread.Sleep` statement. Four participants said they would use an intelligent waiting library (e.g., Awaitility[9]). Three participants would make the request asynchronous (e.g., mock). One participant would separate it into a method with a test step in a more indicative place. Besides, other participants suggested ordering the tests' execution, adding tests containing `thread.Sleep` command at the end of the test suite.

- **UT.** All participants said they would refactor a test method without assertions by including an assertion in the test method. They also reported removing this test method depending on its purpose (or lack thereof).

- **TRW.** Although all participants agreed that this practice could impact the test code quality, only five agreed they should refactor it. Three participants suggested ordering the execution of the test methods; three suggested using *JUnit's* annotations (*@before* and *@after*) and declaring the feature in the setup. One participant mentioned using the *JUnit TemporaryFolder* rule to create specific files for each test method. One participant would refactor it by splitting the class into other classes to contemplate each functionality in a different class. Furthermore, one participant suggested refactoring by grouping the methods into a single test method.

5.4.2.5 Quality improvement strategies (RQ$_5$)

The participants reported that it is necessary to evaluate the test code quality. Each one argued positive points in evaluating test code quality. Among the explanations, four believe that the test code should maintain the same quality standard as the production code. Two of them commented that a good test code directly influences good production code. Four of them pointed out that test code should follow good practices because other developers would use it and be easy to maintain. They mentioned other comments about the importance of evaluating test code quality: avoiding impact on test execution time,

[9]⟨https://github.com/awaitility/awaitility⟩

avoiding rework and waste of time to generate new test cases, and it gives more security to the developer when deploying future versions of the system.

According to the suitable strategies to improve test code quality, three participants mentioned that the company's culture significantly influences test code quality. Two participants said that training is essential to disseminate good practices in the workplace. Two participants talked about the practice of code review, in which one or more developers check code quality written by someone else. Two participants reported the need for tools to analyze the code from the perspective of readability and maintenance. Other strategies were also mentioned: refactoring the test code frequently; improving the test pyramid (unit tests at the bottom of the pyramid, integration test in the middle of the pyramid, and graphical interface test at the top), enhancing the test code quality as well as the production code, and define a standard to be followed within the company.

Furthermore, two participants reported they should write tests with low complexity and coupling. For two participants, tests should be easy to read. One participant mentioned other strategies related to the test structure: simplifying the test to be more objective, writing test methods with few lines, writing clean tests, avoiding test code that does not verify anything, avoiding ambiguous test code, optimizing the test execution time, improving test parallelization and component reuse.

5.5 DISCUSSION

In this Section, we discuss the results obtained and show the main findings of each research question.

5.5.1 Test code creation and maintenance (RQ$_1$)

Most participants reported that they create and maintain testing manually. The tests are maintained at least once, the most frequently reported frequencies were "sometimes" (37.9%) and "often" (26.4%). This finding reinforces the importance of good practices during the implementation of the test code since a test code may not be maintained by the same developer who created it. The easier the readability of the test code, the easier it will be for other developers to understand and make the required modifications correctly.

When asked about automatic test code generation tools, some participants reported tools that do not generate test code. Some participants have confused automated test code generation with tools that support test creation/maintenance. Only 6 participants reported tools that automatically generate test code: EvoSuite (3), Randoop (2), and Diffblack (1). In addition, we also noticed that although some participants stated that they use “Automated” or “Manual and Automated” strategies, they stated that they do not use any tools. Therefore, we had some divergent responses.

Since most participants reported manually creating and maintaining the test code, we already expected to reuse the test code from previous releases. Testing maintenance occurs after updating a feature. Therefore, we can also say that tests tend to evolve together with the software as a whole.

RQ_1 Findings: Developers strongly prefer creating and maintaining test code manually. When they decide to employ automated testing, they manually implement the test code instead of counting on automated test generators.

5.5.2 Awareness about test smells (RQ_2)

As some participants stated that they had heard about test smells but did not know any test smells. Other participants reported that they had never heard of test smells but knew some types. We classified the responses of the participants into four categories: 25.29% of the participants have never heard about test smells, neither know any test smell; 5.75% have never heard about test smells, but know some test smell; 5.75% have heard about test smells, but do not know any test smell; 63.22% have heard about test smells and know some test smell.

We believe that although some participants have already heard about “test smells”, they do not know their types in depth. Possibly, the 5.75% of the participants may have correlated “test smell” with “code smell”. On the other hand, participants may know the types of test smells, but they do not know the term definition due to self-explicit nomenclatures.

Most survey and interview participants believe the test smells explored in this study affect the test code quality, except for the test smell EH. According to the opinion of the survey participants, the test smells that most affect the test code quality are UT, CTL, AR, and TRW. In contrast, the interview results reveal 8 participants believe that UT, TRW, ST, RO, and DA test smells affect test code quality, and only seven agreed that AR, CTL, and EH affect the tests. For this reason, our survey and interview results are similar for the UT and TRW test smells. Further investigation is needed to investigate the test smells AR, CTL, EH, ST, RO, and DA.

Participants believe that test smells affect not only readability and maintenance. It also can have the following consequences: it makes it difficult to reuse the test code; increases the complexity and coupling of the method; generates non-deterministic behavior; it is difficult to identify the root of the problem when a test fails; impacts execution time; and violates important automated testing principles.

Through the analysis, we observed that 49.7% of the participants never insert any of the eight types of test smells presented. 23.4% of the participants rarely ($\cong$ 25%) insert these eight types of test smell into the test code. 14.4% of the participants sometimes ($\cong$ 50%) insert these eight types of test smells into the test code. 9.3% of the participants often answered ($\cong$ 75%) insert these types of test smells, and out of these eight types of test smells, four types of test smells are the often ($\cong$ 75%) inserted test smells, which are: AR, DA, RO, and ST. 3.2% of the participants frequently insert only the test smells AR, DA, EH, RO, and TRW, and of the 5 test smells, only the test smell AR is the most all the time ($\cong$ 100%) inserted. This is in line with the results of other studies (PALOMBA *et al.*, 2016; PERUMA, 2018).

RQ$_2$ Findings: 63.22% of participants ever heard of test smells and know some type of test smells. Most survey and interview participants believe that test smells affect code quality differently. AR is the test smell most inserted in the test suite.

5.5.3 Severity of test smells (RQ$_3$)

Of the 87 participants, 10.8% of the participants consider that eight types of test smells have no degree of severity. 28.7% of the participants consider that all eight types of test smells can be very severe. 24% of the participants consider that 8 test smells can be moderately severe. 15.9% of the participants state that test smells can be considered severe, and 20.5% of the participants consider that test smells can be very severe.

RQ$_3$ Findings: The test smell AR, DA, EH, and RO can be considered a few severe. Test smell ST can be considered moderately severe. The most severe test smells are CLT, UT, and TRW.

5.5.4 How test smells are handled (RQ$_4$)

Seven participants reported the need to refactor the AR, LT, and EH test smells. Some of them suggested refactoring actions. It demonstrates that refactoring test smells is not a trivial task even for practitioners.

All participants agree that there is a need to refactor the following test smells: DA, RO, ST, and UT. All but three participants suggested refactoring actions. About the test smell TRW, only five participants suggested refactoring it.

Among the ways of refactoring the test smells reported by the interviewees, EH, TRW and RO were the test smells with more refactoring alternatives suggested (5 refactoring ways), followed by ST (4 refactoring ways), then CLT (3 refactoring ways), and, with less refactoring suggestions DA and UT (2 refactoring ways). It is worth noting that one of the suggested refactoring forms for UT test smell was unanimous in the interviews. All eight interview participants suggested it, so its refactoring can be general in different scenarios of this type of test smell.

There is no generic refactoring for all types of test smells. Thus, a given type of test smell can be refactored using different techniques. It is essential to evaluate the context of the method and the business domain in each case.

RQ$_4$ Findings: 83.3% of the participants suggested refactoring the eight test smells investigated in this study. The types of test smell present various refactoring alternatives, suitable to different scenarios and domains of these occurrences. Among the 8 test smells presented, the stated refactoring possibilities ranged from 2 to 5 refactoring options.

5.5.5 Quality improvement strategies (RQ$_5$)

All the interviewed participants believe in the relevance of improving test code quality, but they commonly neglected it. The interviewees reported that they should give the same importance to the test code as the production code.

Furthermore, the interview results show that company culture and training can be great allies in the quest for test code quality. We believe that when a company incorporates best practice principles, the pursuit of test quality becomes more natural. There can be dissemination knowledge among the software development and testing team, providing the dissemination of good practices. We also realize that some practical concepts, such as the single responsibility principle and clean code test, can avoid more than one test smell.

RQ$_5$ Findings: Strategies such as organizational culture, training, and best practices can be adopted to facilitate understanding and maintenance of the test code.

5.6 THREATS TO VALIDITY

Construct validity. We limited the target audience of our survey to the Brazilian scenario. Therefore, this sample may not be generalized to testers from around the world. However, we sought to diversify the sample with different software tester profiles. There were participants from all regions of Brazil, with different profiles, positions, levels of experience, education, and different companies, bringing more breadth to perspectives reported by these practitioners. Due to the number of questions in the survey, participants may not have answered the questions adequately. To reduce the threat to validity, we grouped the questions into specific sections to better target the questions and answers.

Internal validity. Some questions may not have been clear to some participants, misinterpretations may have occurred during responses and when analyzing them. To reduce these situations, we conducted pilot studies before initializing the surveys and interviews. In addition, the analyses were done by more than one person, seeking consensus in understanding the participants' perceptions.

External validity. Our analysis was based on the results of a survey of 87 participants and 8 interviews. We select software testing practitioners with experience with the JUnit framework. But, a more significant number of participants would bring more accurate results. The knowledge, experience, interpretations, and timing of these participants may interfere with the discussions and conclusions of this paper. To mitigate this, we filtered out discrepant responses through selection criteria.

Conclusion validity. In our study, we use real code examples taken from GitHub open repositories. For the same type of test smells, there are more critical cases than others, so the examples presented in the survey to represent test smells may influence the participants' perception. To mitigate this problem, we informed that we used code snippets presented only to exemplify similar practices, and our questions were directed to the practice, not for that specific code snippet.

5.7 CONCLUSION

This chapter presented a study we carried out to investigate the effects of test smells on maintenance and test code quality from the practitioners' perspective and experience in software testing. We gathered data from 87 practitioners and interviewed eight with expertise in software testing. For each type of test smell, they presented a different severity level. We realize that each developer views test smell practices differently concerning the severity and has their own choices on handling them. According to the developers' perspectives, the types of test smells vary greatly in different scenarios. Most developers recognize potential problems and impacts of test smells, but the complexity of treatment and refactorings may vary case by case.

The results show that 72.41% of the practitioners prefer to create and maintain test code manually. 63.22% of the practitioners know some test smells. Of the 8 types of test smells investigated, 3 test smells (AR, DA, and RO) are frequently inserted into the test cases. According to the severity analysis, the participants considered CLT, UT, and TRW to be very severe. Although the test smells have different severity levels, 83.3% of the participants agree with refactoring the 8 test smells investigated. Moreover, the participants present several alternatives to refactoring the test smell for different scenarios and domains of these occurrences.

As future work directions, it would be interesting to investigate the testers' perspective on other programming languages with other testing frameworks and examine whether the behavior of test smells is applicable in different domains.

Chapter 6

ON THE INFLUENCE OF DEVELOPERS' EXPERIENCE ON SOFTWARE TEST CODE QUALITY

This Chapter presents the third empirical study we carried out, intending to answer the **RQ3** of this thesis. This study presents the following contributions: (i) an analysis of the experience and practices of core and peripheral developers during software maintenance and evolution; (ii) an analysis of which type of developers, either core or peripheral, insert and remove more test smells; and (iii) motivation to make developers aware of the importance of the maintenance and refactoring test code to improve its quality (CAMPOS; MARTINS; MACHADO, 2022).

We structured this chapter as follows: Section 6.1 presents the motivation, research problem, and objective. Section 6.2 presents the proposed research methodology for this study. Section 6.3 presents the answers to the research questions. Section 6.4 discusses the main findings. Section 6.5 discusses the threats to validity. Section 6.6 concludes the Chapter.

6.1 INTRODUCTION

In this study, we aim to investigate the relationship between the developers' experience concerning their contribution to the test code and the test code quality from the perspective of test smells. We performed an empirical study with four projects to analyze the insertion and removal of test smells by experienced developers compared to the less experienced ones. Only four projects were selected because, during the analysis, a recurring pattern was observed in the collection. First, we selected open-source Java projects using the *SEART tool.* Second, we identified test smells in the four open-source projects through manual and automated approaches to identify the authorship of the test smells. Third, we analyzed the data to investigate how the developers' experience influences the insertion and removal of test smells, how long the test smells survive, and when the test smells are removed from open-source projects. Fourth, we performed the analysis of the data found. We define three research questions (RQ) to guide our study :

RQ1. How does the developer's experience in OSS projects affect the quality of test cases? This RQ aims to analyze the relationship between the developers' experience and the quality of the implemented test code regarding the presence of test smells. We measure the developer's experience by the ratio of the number of commits performed by the developer.

RQ2. Does the developer's experience in OSS projects influence the insertion or removal of test smells? This RQ aims to unveil if developers with more experience insert or remove fewer test smells.

RQ3. What are the moments that core or peripheral developers insert or remove test smells? In this RQ, we aim to identify when developers insert or remove test smells from the projects during their evolution.

6.2 RESEARCH METHODOLOGY

Our study design encompasses four steps, as Figure 6.1 shows: *Project selection*, *Detection of test smells*, *Data collection* (e.g.; detection of test smells and refactoring operations, developer experience, and survivability), and *Analysis of the results.*
Step 1 - Selection of Java projects - We used the *SEART tool* to select open-source Java projects by applying the following criteria: i) repositories with at least ten stars, ii) projects implemented in Java, iii) non-forked projects, iv) mature projects – we applied the filters to select projects that continuously receive community support: their longevity is equal to or greater than five years, and they have at least five years of releases–, v) projects containing at least ten tags, and vi) projects containing at least ten test classes. We retrieved 55 projects using *SEART* tool. The next step was to analyze them with the support of the *JNose Test* tool. Only 10 out of 55 projects follow the JUnit naming conventions required to execute the tool. We randomly selected four projects from this set of selected projects to perform a preliminary study. All these projects had a set of test cases written in JUnit4.

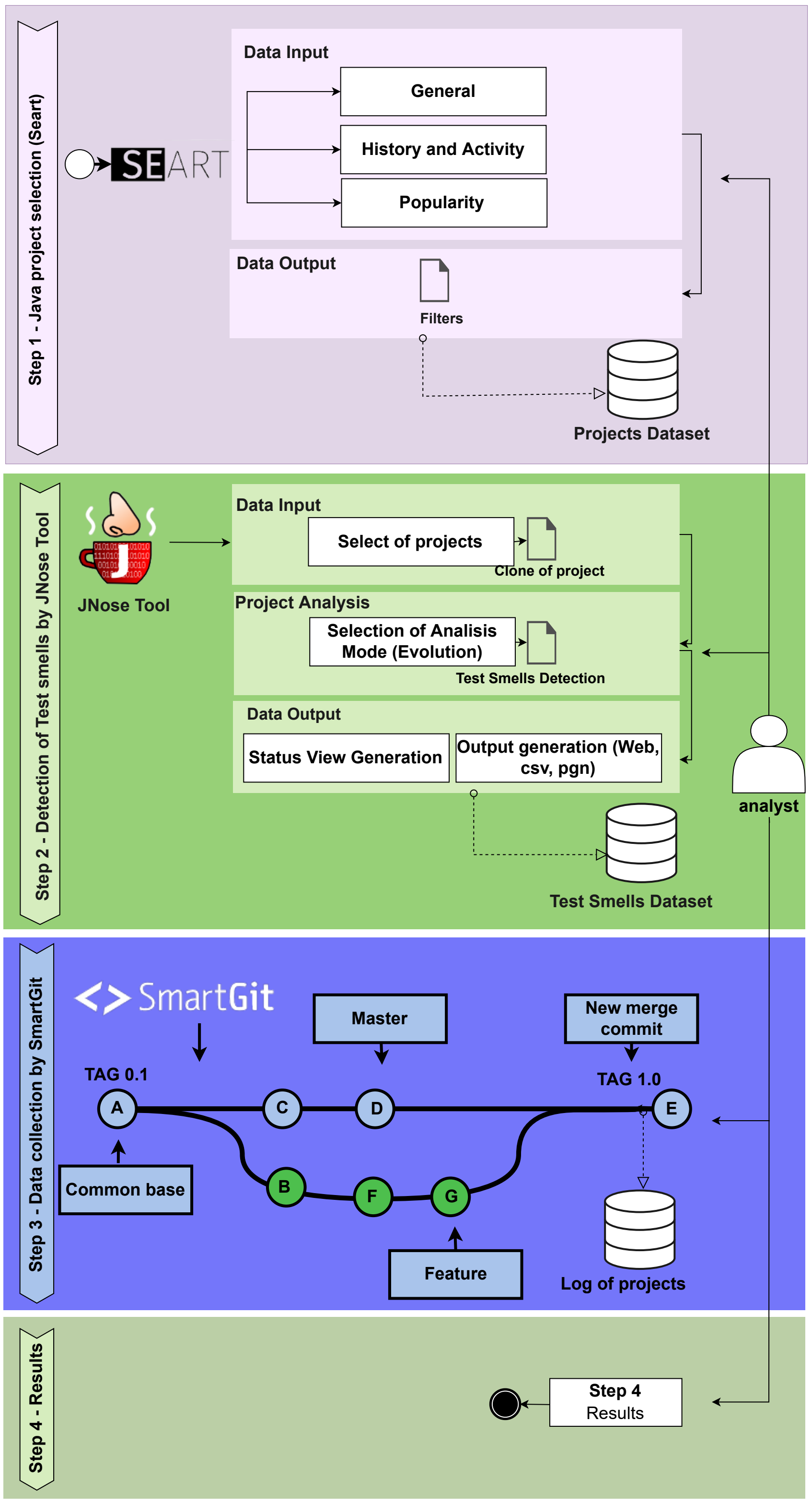


Figure 6.1: Study Design

Step 2. ***Detection of test smell by JNose Test*** - The result of phase 1 is the input to the JNose Test. In Step 2, we used the *JNose Test* to generate the list of test smells present in the selected projects. The *JNose Test* creates an abstract syntax tree and performs static analysis of the test code to detect test smells. As a result, the tool generates different *.csv* files; a *csv* for the test smells detected in the last commit and another for the test smells detected in each project version.
Step 3. ***Data collection through project logs by SmartGit Tool*** - This step's inputs are the projects selected in phase 1, and the result of the test smells present in the projects in step 2. In step 3, we used the SmartGit tool to generate the project logs, and through the logs, we collected some metadata, such as the author of the commit, the set of commits composing a tag, and the set of detected test smells. The logs provided by the SmartGit tool range from the creation of the first commit to the last commit in all branches, i.e., the complete commit history of the projects.
Step 4 - Analysis of the result - At this phase, we analyzed the data to present the results of our empirical study. We followed an exploratory and quantitative analysis to investigate the research questions. As a result, we present the developers' experience influencing the insertion and removal of test smells during the project's life cycle.

6.2.1 Classification of core and peripheral developers

In this section, we present the definitions of the developer's roles and experience. The developer's experience in the project is defined based on the number of commits performed by each developer (TERCEIRO; RIOS; CHAVEZ, 2010; CALEFATO *et al.*, 2022). We compute a threshold at the 80% percentile and separate the developers who have a commit count above the threshold as core developers and below the threshold as peripheral developers.

The smaller groups are the *core* developers of the project, ranging from 4 to 30 developers per project, whose developers do most of the work (TERCEIRO; RIOS; CHAVEZ, 2010). The larger group of *peripheral* developers contribute more directly but less frequently, and range from 15 to 105 developers per project. These developers work directly on bug fixes, documentation, etc. (TERCEIRO; RIOS; CHAVEZ, 2010). Peripheral developers are not permanent members of the core development team and work voluntarily, dedicating their time and talent to improve the product quality (SETIA *et al.*, 2012).

Core developers tend to accomplish most of the activities. They have greater decision-making authority, while *peripheral* developers accomplish a smaller portion of the activities and have no decision-making authority. In addition, the core and peripheral developers exhibit different behaviors related to the discussion about the project (TERCEIRO; RIOS; CHAVEZ, 2010).

6.2.2 Data Collection

We collected data from the first to the latest commit modifying test classes in all branches of four open-source Java projects, as follows:

- The project *airbnb/lottie-android*[1] is a mobile library for Android and iOS that parses Adobe After Effects animations exported as JSON with Bodymovin and renders them natively on mobile.
- The project *code4craft/webmagic*[2] is a simple crawler for Java developers. *WebMagic* contains two parts: *Webmagic-core* is a simple and modularized implementation of crawler, and *webmagic-extension* supply some convenient features for crawler developing.
- The project *hs-web/hsweb-framework*[3] is a fully responsive background management framework based on spring boot.
- The project *apolloconfig/apollo*[4] is a configuration management system for different applications and different clusters. It is suitable for microservice configuration management scenarios.

We use two tools for the data collection, *SmartGit* and *JNose Test*. The former presents the project history through a friendly interface, from which we trace all the projects' commits and authorship. It is worth noting that the *JNose Test* relies on the JUnit naming convention to identify the test classes, adding the word 'Test' as either prefix or suffix to the production class name. For example, if the production class's name is *Job.java*, then the test class's name would be *JobTest.java* or *TestJob.java*. Therefore, we only analyzed the test classes that follow the JUnit naming conventions.

In addition, the projects can have modules, and we analyze them individually. For example, the *hs-web/hsweb-framework* has the *hs-web/hsweb-authorization* and *hs-web/hsweb-core* modules. The project and modules can have test classes with the same name

[1]⟨https://github.com/airbnb/lottie-android⟩
[2]⟨https://github.com/code4craft/webmagic⟩
[3]⟨https://github.com/hs-web/hsweb-framework⟩
[4]⟨https://github.com/apolloconfig/apollo⟩

but different test methods and smells; therefore, we count the test smells instances for each.

After detecting the test smells with the *JNose Test*, we used the *SmartGit* to search for the author and date of the commits containing test smells. We used the “last change takes it all” strategy to assign all the knowledge in a test file to the last developer who modified it (FERREIRA; VALENTE; FERREIRA, 2017). Therefore, we manually analyzed the results of both tools *JNose Test* and *SmartGit* to establish the authorship of a given test smell. Besides that, our manual analysis complemented the data collection with the *JNose Test* regarding two test smells: *AR* and *IT*. Suppose a test method contains two assertions without an explanation message (*AR*), and the assertions have the same parameters (*DA*) or two numeric literals as parameters (*MNT*). In that case, the *JNose Test* will only count one of the assertions as one *AR*. If the test method has two *AR*, two *DA* or two *MNT*, even though *JNose Test* only detects one *AR* or *DA* or *MNT*. In the manual analysis, we count each instance of *AR* or *DA* or *MNT*. In addition, the *JNose Test* does not detect test smells after the *@Ignore* tag. We annotate all test smells in the test class during the manual analysis, even if the test class is ignored.

6.3 RESULTS

We analyzed the evolution of all the four projects mentioned above. For transparency, we made all raw data from the analysis on the Zenodo platform available.[5]

Table 6.1 presents the characterization of the selected projects in terms of the number of production classes currently in the project (NPC), the number of test classes in the latest version (NTC), the number of analyzed test classes during projects’ evolution (NATC), the number of test classes removed during the projects’ evolution (NDTC), and the ratio of production classes under test classes (PCTC), considering the NPC and NTC metrics.

6.3.1 Developer’s experience and test case quality (RQ1)

We analyzed all commits in test classes to identify the authorship of the smelly test files and the number of test smells in the test files. Then, we classified the developers into core and peripheral. While analyzing the project data, we observed that each tag could have one to four core developers. The number of core developers is smaller than the peripheral developers. For example, the *hsweb-framework* project has four core developers

[5]⟨https://doi.org/10.5281/zenodo.7110141⟩

Table 6.1: Summary of selected projects

Projects	NPC	NTC	NATC	NDTC	PCTC
hs-web/hsweb-framework	599	91	123	67	15.19%
code4craft/webmagic	223	83	56	16	37.22%
airbnb/lottie-android	267	27	21	8	10.11%
apolloconfig/apollo	712	418	186	18	58.71%
Total	1,801	619	386	109	34.37%

Legend: Number of Production Classes (NPC), Number of Test Classes in the Latest Version (NTC), Number of Analyzed Test Classes (NATC) and Number of Deleted Test Classes (NDTC), Percentage of Production Classes under Test Classes (PCTC)

who performed 2,260 commits and 15 peripheral developers who performed 24 commits. However, only one core developer stood out in most tags of the *hsweb-framework* project. Similarly, other projects have a small number of core developers. Only two or three of these developers perform commits in almost every release of projects.

Figure 6.2 shows the number of commits related to the test classes per core and peripheral developers in the four projects. The *hs-web/hsweb-framework* project comprises four core developers. Of these four core developers, only one has performed a total of 2205 commits, while the others have performed only 55.

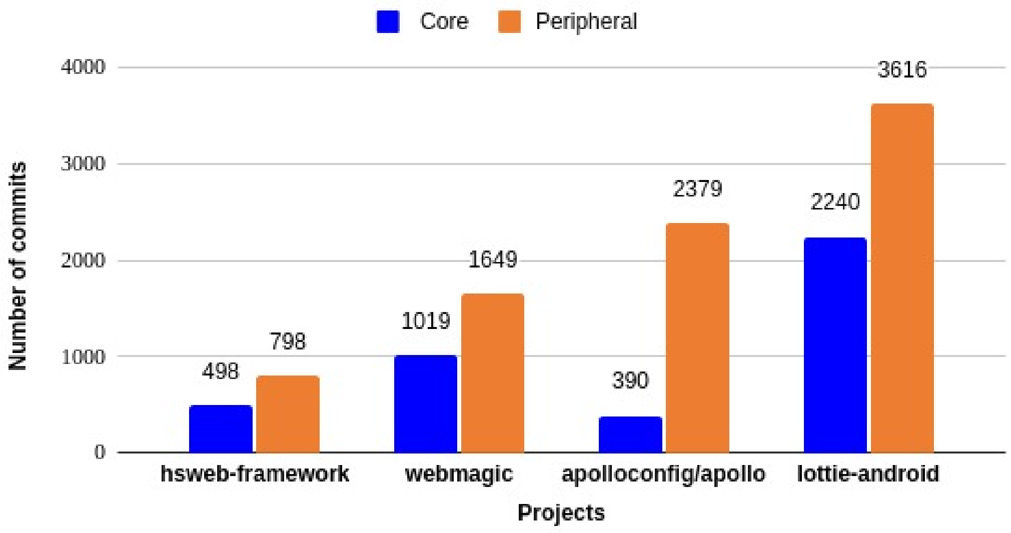


Figure 6.2: Number of commits in test classes of each project by developers' role

Figure 6.3 shows the authorship of test smells per core and peripheral developers in the four projects. In the project *apolloconfig/apollo*, the peripheral developers inserted 519 test smells, while the core developers inserted 3,046 test smells. In the same project, the core developers performed most of the commits and inserted or removed more test smells. Some developers play both roles in larger projects like *apolloconfig/apollo*. In some releases, a specific developer is a core developer, while in others, the same developer is a peripheral one. In the smaller projects, core developers contribute more than peripheral developers to the test code and consequently could insert and remove more test smells than peripheral ones.

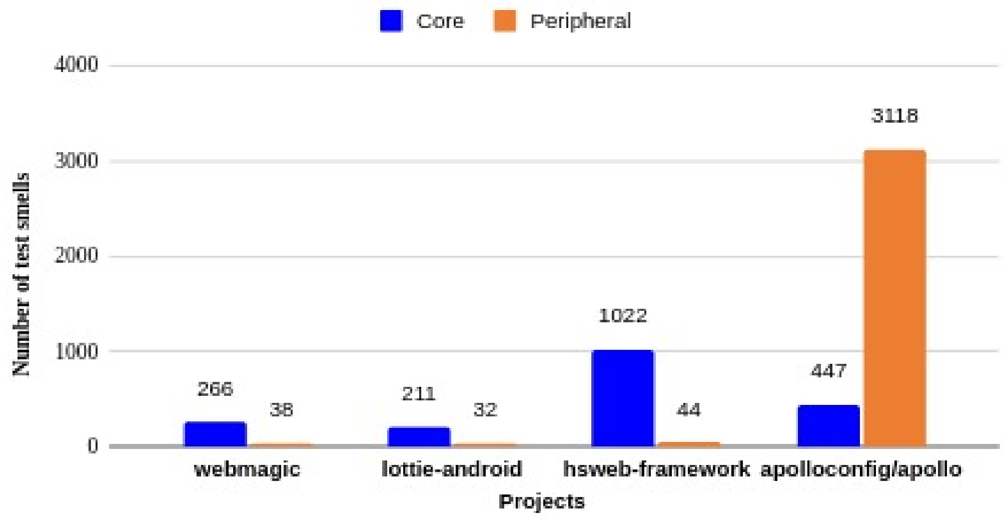


Figure 6.3: Number of test smells inserted by developers' role

We analyzed 386 test classes from the four projects, as follows: 123 test classes in the *hs-web/hsweb-framework* project; 56 test classes in the *code4craft/webmagic* project; 21 test classes in the *airbnb/lottie-android* project; and 186 test classes in the *apollo-config/apollo* project. The results indicate that the core developers were responsible for inserting 4,604 out of 5,178 (88.91%) test smells and removing 971 out of 1,081 (89,82%) test smells. In contrast, the peripheral developers inserted 574 (11.09%) and removed 110 (10.18%) of the test smells.

Table 6.2 presents the number of test smells in each project. We detected test smells using the *JNose Test* and complemented it with a manual analysis of the modified test classes. The *JNose Test* column presents the number of test smells detected using the *JNose Test*, and the *Manual* column presents the number of test smells identified during the manual analysis. The *Total* column presents the total test smells detected in each project (i.e., the sum of test smells detected through manual and automated analysis). The *JNose Test* detects the test smells present in the last commit of a tag. For example, a tag named *pre-release* contains 150 commits, from c_1 to c_{150}. In the tag *pre-release*, the *JNose Test* detects only the test smells present in c_{150}. However, the test code may

Table 6.2: Test Smells Detected

Project	JNose Test	Manual	Total
lottie-android	189	54	243
hsweb-framework	717	349	1,066
webmagic	260	44	304
apollo	1,827	1,738	3,565
Total	2,993	2,185	5,178

contain test smells inserted or removed between the commits c_1 to c_{149}. Therefore, we manually added the test smells that the *JNose Test* did not detect for a tag.

After executing the JNose tool, we manually analyzed the test code to trace the exact location of a test smell in the current and parent commit and determine whether the modification introduced or removed the test smell. Such functionality is not available in any of the existing test smells detection tools (ALJEDAANI *et al.*, 2021). In addition, we also needed to track whether consecutive tags kept the same test smell to remove it from our analysis; otherwise, we can have duplicated test smells through the project's evolution.

6.3.2 Relationship between test smell insertion/ removal and developer's experience (RQ2)

We collected the test smells in each project and classified the developers who inserted or removed the test smells as core or peripheral developers. We noticed that only one developer performs most of the commits that compose a tag, i.e., only one developer is responsible for the entire project.

Figure 6.4 shows the number of test smells inserted and removed on all projects. The core developers inserted and removed more test smells than the peripheral ones. The core developer who has worked on all tags of the project is the developer who has obtained the highest number of commits performed relative to the other peripheral developers in the project. Active and regular contributors like core developers, even when working as peripheral developers, are expected to develop and test more features and are prone to insert more test smells.

Figure 6.5 presents the number of test smells inserted by the core and peripheral developers. We analyzed all 18 test smells types.which the *JNose Test* detects. However, the projects contain different types of test smells. Specifically, the *airbnb/lottie-android* has 9 types, *code4craft/webmagic* has 15 types and *hs-web/hsweb-framework* has 16 types,

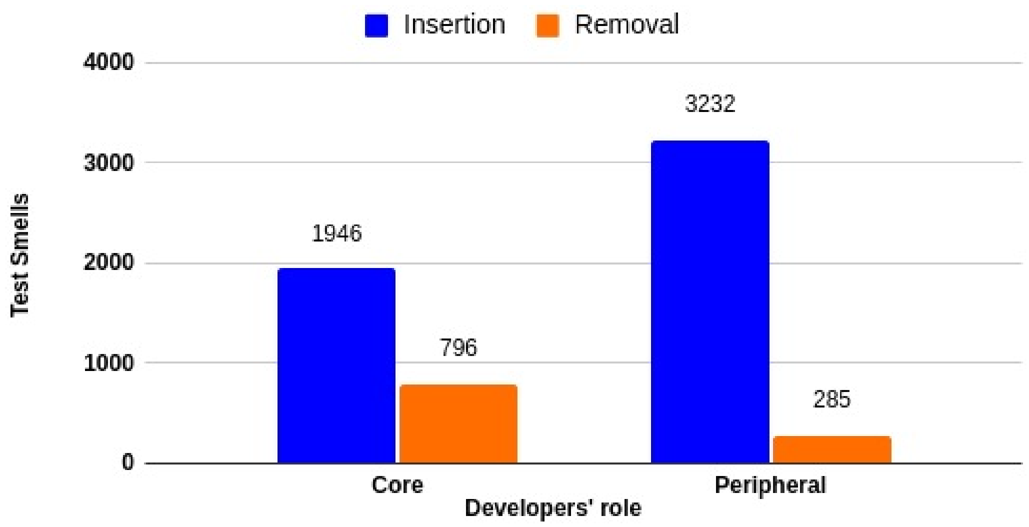


Figure 6.4: Number of insertions and removals by developers

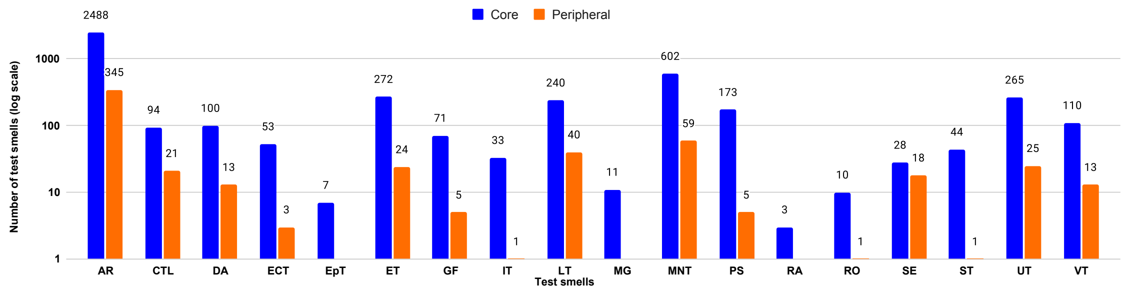


Figure 6.5: Number of test smells inserted by developers

and *apolloconfig/apollo* has 17 types of test smells. The test smells common to the four projects are: *AR*, *CTL*, *ET*, *ECT*, *IT*, *LT*, *UT* and *MNT*. The core developers frequently inserted the test smells: *AR* with 2,488 instances, *MNT* with 602 instances, *LT* with 240 instances, and *UT* with 265 instances.

To analyze the test smell removal, we followed the same steps to analyze the test smell insertion. Figure 6.6 presents 17 types of test smells removed from the test code by the core and peripheral developers. The core developers are the ones that remove more test smells. The test smells frequently removed are: *AR* with 471 instances, *MNT* with 160 instances, and *PS* with 131 instances removed. The most inserted test smells are also the most removed by both developers. Among the 2,833 *AR* inserted, only 471 were removed, 661 *MNT* were inserted, and only 160 were removed, 296 *ET* were inserted, and only 48 were removed.

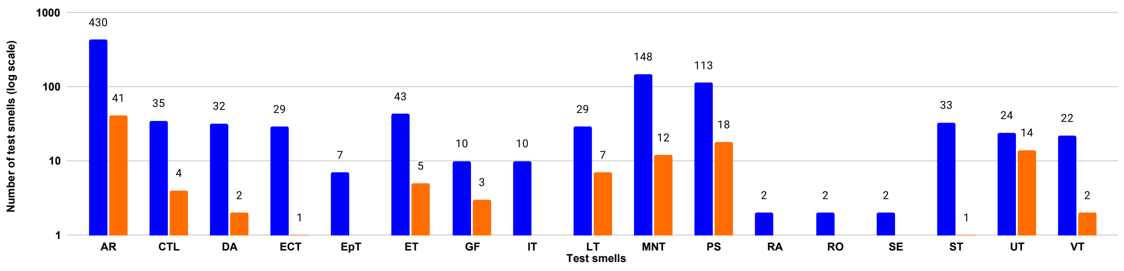


Figure 6.6: Number of test smells removed by developers

6.3.3 Moment when developers to insert or remove test smells (RQ3)

In this study, we also analyzed the moment of test smell insertion. We observed that of the 5,178 test smells inserted, 3,484 (67.28%) test smells were inserted during test class creation, 1,152 (22.25%) test smells were inserted during method creation, and 542 (10.47%) test smells were inserted during test case refactoring, as Figure 6.7 shows. We can see that the *AR*, *MNT*, *PS*, *ECT*, and *CTL* are the most frequently identified test smells inserted during the creation of the test classes, test methods, and their refactorings.

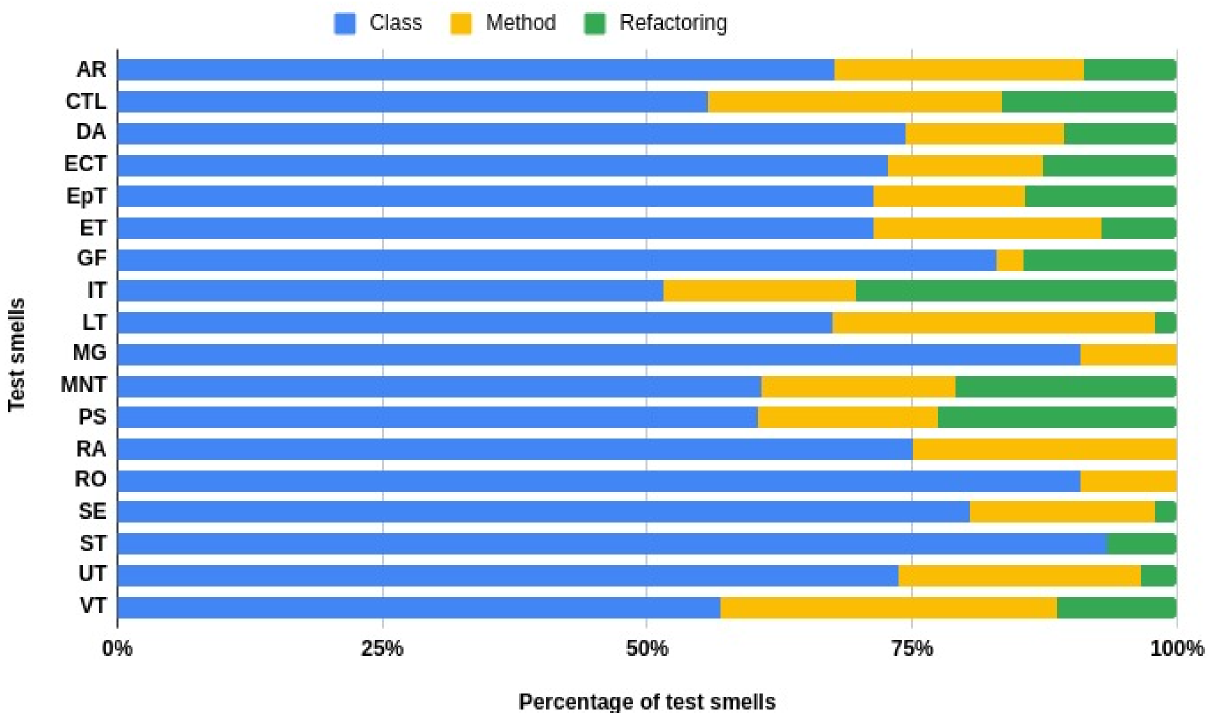


Figure 6.7: Insertion of test smells when deleting the test class, deleting test methods, or refactorings.

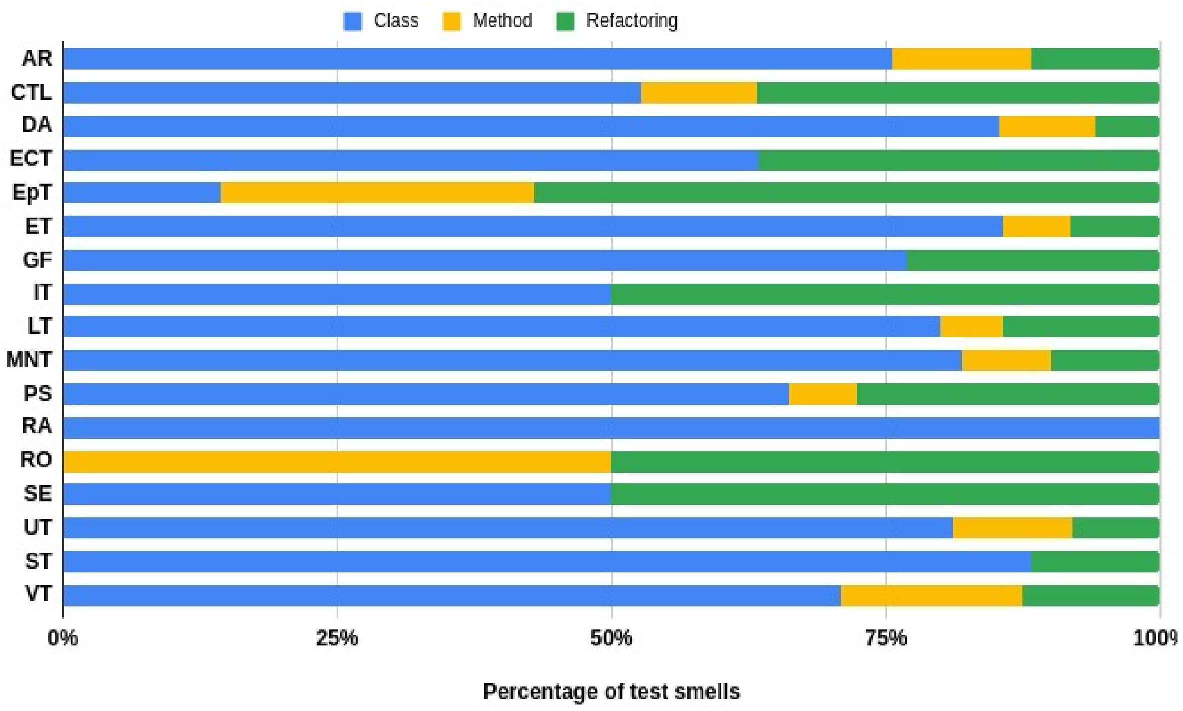


Figure 6.8: Removal of test smells when deleting the test class, deleting test methods or refactorings.

Figure 6.8 shows the common refactorings and removal test smells. Of the 1,081 removed test smells, 805 (74.47%) were removed after removing the test class, 103 (9.53%) were removed due to removing the test method, 167 (15.45%) were removed due to refactoring the test cases, and 6 (0.56%) were due to the reallocation of test smells to other methods. Refactorings refer to any changes to the test code during the software evolution (e.g., adding or removing lines or libraries). The *AR*, *MNT*, and *PS* test smells are removed frequently.

The *code4craft/webmagic* project has 19 tags, the *hs-web/hsweb-framework* project has 18 tags, the *airbnb/lottie-android* has 13 tags, and *apolloconfig/apollo* has 34 tags. The number of test smells inserted or removed varies during the projects' evolution. Next, we characterize the projects' evolution concerning the test code quality, particularly from the perspective of test smells.

Figure 6.9 presents the analysis of the *code4craft/webmagic* project. There was an insertion of 15 types of test smells during the project's evolution. In this project, 304 test smells were inserted; 46 (15.13%) of test smells were inserted in the first tag of the project, and then the insertion of test smells increased in each tag. The most frequently inserted test smell was the AR, reaching up to 16 insertions per tag.

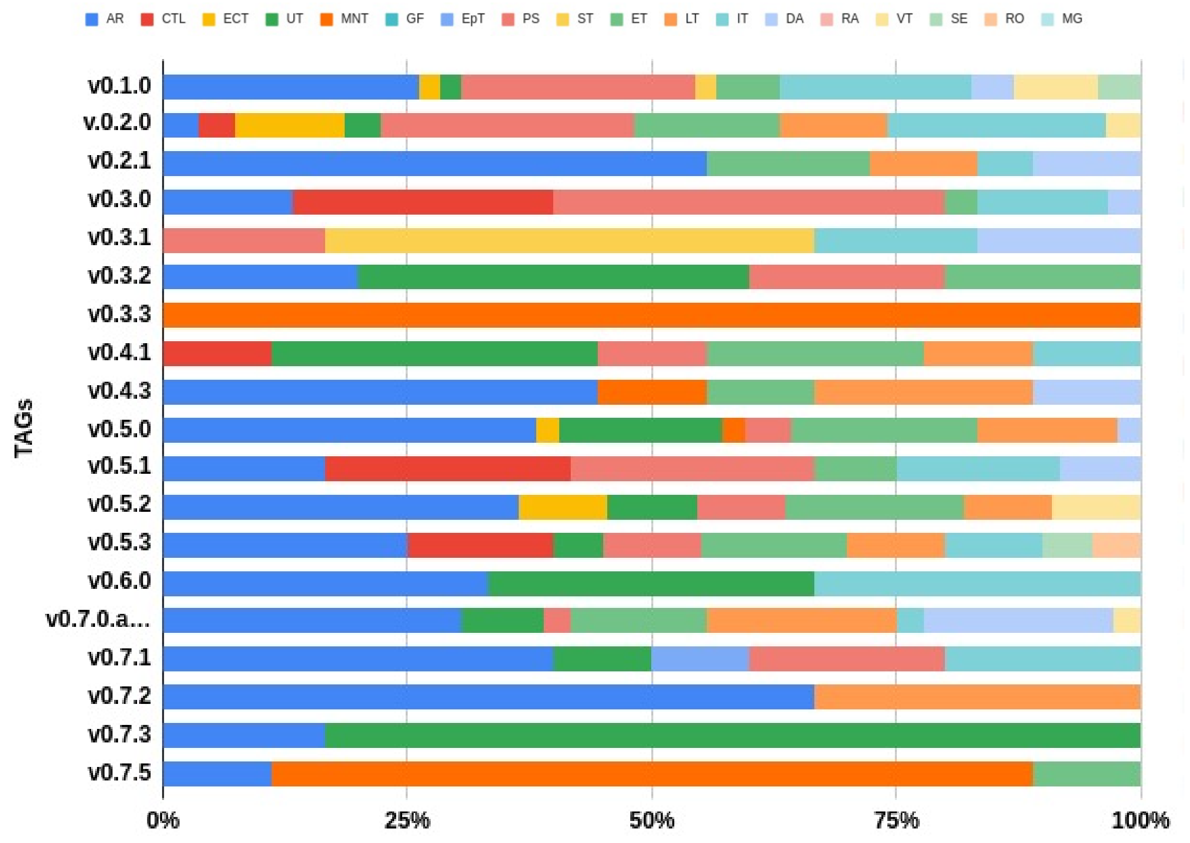


Figure 6.9: Evolution of the *code4craft/webmagic* project

Figure 6.10 presents the evolution of the *hs-web/hsweb-framework* project, which has 16 types of test smells. In this project, 1,066 test smells were inserted, the project has an insertion of 599 (56.19%) of test smells from the second project's tag. Then, the test smells are introduced gradually over time. In this project, AR is the most inserted test smell during software evolution. Only the UT was inserted in the last project's tag. The test smell with the fewest insertions was the GF with only one insertion during the entire software evolution.

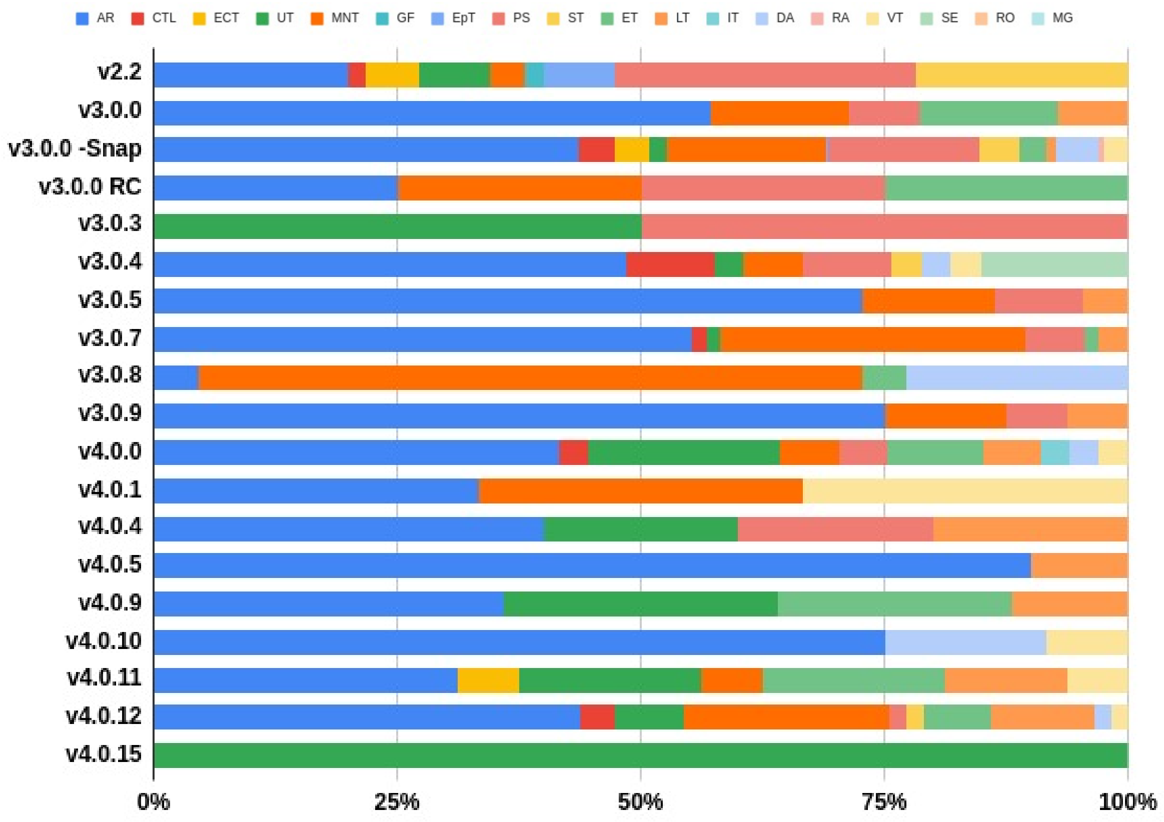


Figure 6.10: Evolution of the *hs-web/hsweb-framework* project

Figure 6.11 presents the evolution of the *airbnb/lottie-android* project, which has 9 types of test smells. Compared with the other two projects, *airbnb/lottie-android* is small but presents diverse test smells. In this project, 243 test smells were inserted. In the first tag, there was an insertion of 26 (10.70%) test smells, and it increases in the third tag with the insertion of 50 (20.58%) test smells, and in the sixth tag with the insertion of 64 (26.34%) test smells. In the last tag of the project occurs the insertion of only two types of test smells (MNT and AR). In this project, the insertion of the test smells UT, GF, ECT, and IT occur in one tag, and the insertion of CTL occurs in two tags.

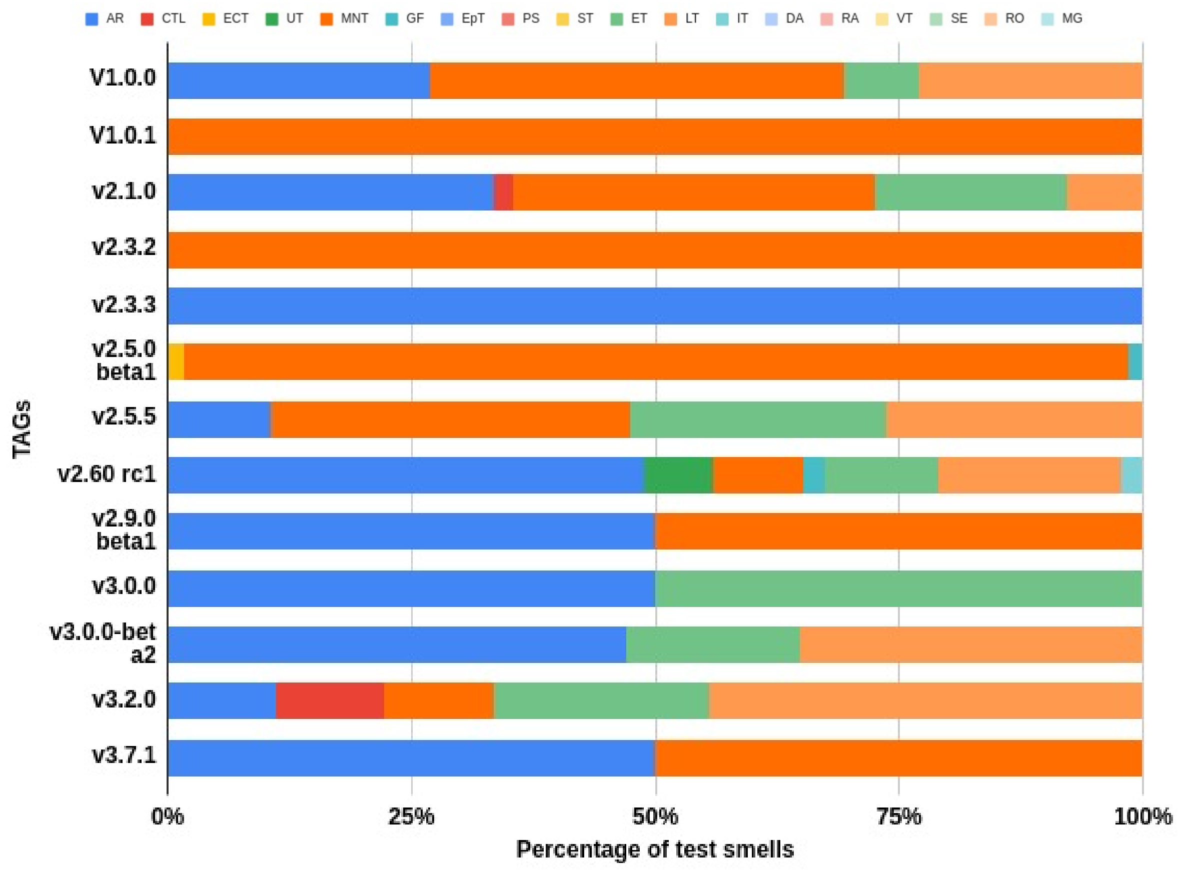


Figure 6.11: Evolution of the *airbnb/lottie-android* project

Figure 6.12 presents the evolution of the *apolloconfig/apollo* project, which has 17 types of test smells. This is the largest project analyzed in this study and contains the most significant number of peripheral developers. However, three core developers have performed the most significant number of commits in all project tags. In this project, 3,565 test smells were inserted. In the first tag, there was an insertion of 1,608 (45.11%) of test smells, and it increases in the sixth tag with the insertion of 242 (6.79%) test smells, and in the thirteenth tag with the insertion of 198 (5.55%) test smells.

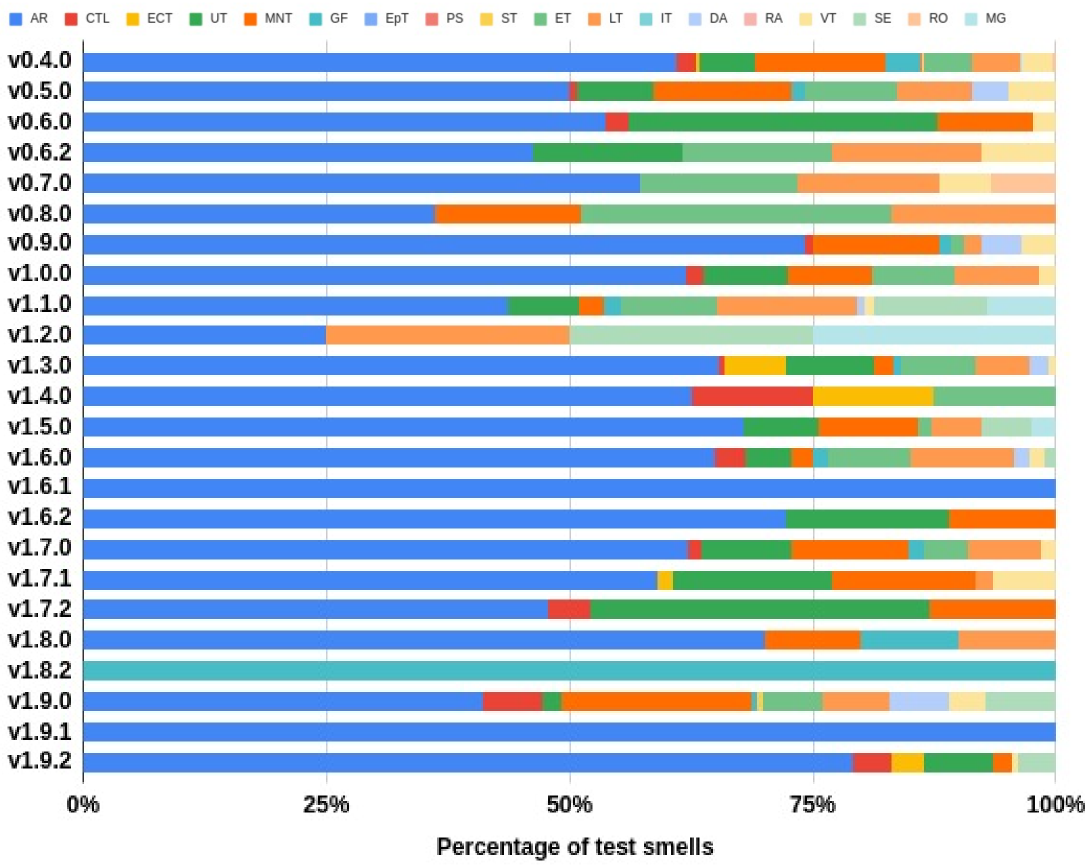


Figure 6.12: Evolution of the *apolloconfig/apollo* project

6.4 DISCUSSION

This section discusses the results achieved and presents the main findings for each research question.

Developer's experience and the quality of test cases. Commonly, open-source projects have few core developers and several peripheral developers. On the one hand, when analyzing a small project like the *hsweb-framework*, we could observe that the core developer performed a total of 2,260 commits and inserted 1,052 test smells. Conversely, all the peripheral developers performed 24 commits and inserted 14 test smells. On the other hand, in a larger project like the *apolloconfig/apollo*, the core developers performed 2,579 commits and inserted 3,046 test smells. In this project, three core developers were the most active in all versions, in certain versions one of these three core became the peripherals developer.

Each project has one developer responsible for creating and maintaining most production and test classes. This can overwhelm a particular developer, who would likely prioritize developing new features or maintaining existing ones rather than testing them. As a result, most production classes do not have their respective test classes, which may harm software quality. For example, only 15.19% of the 599 production classes in the *hsweb-framework* project have been tested.

Developer's experience and test smell insertion/removal. The data analysis shows that, from a total of 5,178 test smells, the core developers were responsible for inserting 4,604 (88.91%) test smells, while peripheral developers were responsible for inserting 574 (11,09%) ones. Regarding the removal of 1,081 test smells, core developers were responsible for removing 971 (89.82%) test smells, while peripheral developers removed 110 (10.18%) test smells.

The *airbnb/lottie-android* core developer is responsible for 1,030 (94.93%) of the commits performed, the *code4craft/webmagic* core developer is responsible for 861 (94.93%) of the commits performed, and the *hsweb-framework* core developer is responsible for 2,260 (98.95%) of the commits performed. In the *apolloconfig/apollo* project, the core developer is responsible for 2,579 (95.10%) of the commits performed. In the projects *airbnb/lottie-android*, *code4craft/web-magic* and *hsweb-framework*, since the core developers are overwhelmed, the project may lack test cases. In the *apolloconfig/apollo* project, three core developers are the most active ones, still they may work as peripheral developers in some project's tags. The core developers that contribute to the project may use the production or test code as documentation, and they would analyze the core developers' code patterns and follow them. We might observe an indication that the developer's experience could be directly related to the number of test smells inserted in a project. However, since core developers commonly establish and follow coding practices (e.g., patterns and code style), they might influence the practices used by peripheral developers.

Indeed, there is open room for further investigation. More qualitative studies are necessary to understand whether the coding practices shared between core and peripheral developers might influence test smells' insertion and removal.

Time of test smells insertion and removal. In this study, we bring evidence about the practices performed by core and peripheral developers. Of the 5,178 test smells inserted, 3,484 (67.28%) were inserted during test class creation, 1,152 (22.25%) were inserted during the method creation, and 542 (10.47%) were inserted during test case refactoring. Concerning the test smell removal, 805 (74.47%) test smells were removed as their parent classes were deleted from the project, 103 (9.53%) were removed as the test methods were deleted from the classes, 167 (15.45%) were removed due to test case refactorings, and 6 (0.56%) were due to the reallocation of test smells to other methods.

We observe that the developers insert test smells while creating test classes or methods. Test smells are commonly removed when test classes are removed. Besides, the production classes undergo constant changes, but the test classes are not refactored frequently. This evidence shows that the quality of the software needs to be improved from the beginning of the project's creation, not only during refactorings.

6.5 THREATS TO VALIDITY

Internal validity. The present study only considered open-source, Java, and Git-based repositories. To mitigate this threat, we selected projects that vary in size, contributors, number of commits, and refactoring strategies. We performed data collection from the projects manually. We likely did not capture the data as intended during data collection. Other researchers also assisted in collecting the data manually from the projects to mitigate this threat. GHTorrent is a database for collecting data from the GitHub Application Programming Interface (API) (GOUSIOS; ZAIDMAN, 2014). Our study focused on four projects based on an analysis of the GHTorrent database through the SEART tool. Therefore, the reliability of our work depends in part on the reliability of the GHTorrent database collected through GitHub. Furthermore, the manual analysis was performed by only one person for each repository, presenting a possible risk of unreliable results.

Construct validity. The interpretation of data can influence the research results. The projects have different test smells, and comparing the same test smells in all projects is unfeasible. It is challenging to get developer experience due to the volume of data collected, and the experience can be subjective. To mitigate this threat, we used the number of commits to classify the developers' role (TERCEIRO; RIOS; CHAVEZ, 2010; CALEFATO *et al.*, 2022). In addition, the JNose tool only detects classes following JUnit naming conventions. Therefore, we may have missed some test classes.

External Validity. The results of our study are based on the analysis of four open-source projects. The projects vary in size, amount of test classes, and the number of commits. They are mature and relatively successful projects. We present the methodology of this study in more detail that may allow future replications.

6.6 CONCLUSION

This Chapter presented an empirical study to investigate the relationship between developers' experience and test code quality from the perspective of test smells. We analyzed four open-source Java software projects to collect test smells and their authorship through the projects' evolution.

We found that, on the one hand, core developers are fewer in number, but they are responsible for developing and removing production classes and test classes from the project. On the other hand, peripheral developers are the highest in number, but they often only perform bug fixes. Regarding the test code quality, the developers inserted 5,178 test smells and removed 1,081 (20.88%) over time. In addition, the ratio of production classes under test is low, likely due to the overload on the core developers. Despite the overload of the core developers, the projects have a continuous maintenance and evolution process.

We could observe that developers commonly refactor test smells to remove specific statements, e.g., they may use different assertion styles or test frameworks to remove test smells. Test frameworks have different syntaxes, and the lack of experience and knowledge may lead developers to implement more detailed assertions. In addition, developers face difficulties performing refactoring tasks, as only 15% of test smells were removed through test case refactoring. Future research should attempt to understand developers' difficulties in refactoring test smells.

Chapter
7

DEVELOPER CONTRIBUTIONS TO TEST SMELL SURVIVABILITY: A STUDY OF OSS PROJECTS

This Chapter presents the empirical study we carried out to answer the **RQ4** of this thesis. We investigated test smell survivability and explored the interaction between developer expertise and refactorings across four open-source GitHub projects. Additionally, our study highlights the significance of understanding the dynamics of test smells and emphasizes the need for continuous improvements in test code quality (CAMPOS *et al.*, 2023b).

We structured this chapter as follows: Section 7.1 introduces the empirical study. Section 7.2 presents the study methodology. Section 7.3 presents the results. Section 7.4 discusses the study results. Section 7.5 discusses threats to the validity. Section 7.6 concludes the Chapter.

7.1 INTRODUCTION

Researchers have investigated the developers' roles and their implications for developing and maintaining open-source projects for decades. For instance, researchers have proposed approaches to classify the developers into core and peripheral based on their contributions (CROWSTON *et al.*, 2006; TERCEIRO; RIOS; CHAVEZ, 2010; JOBLIN *et al.*, 2017), investigated the developers' motivation and barriers to joining the projects as contributors (STEINMACHER *et al.*, 2016; LEE; CARVER, 2017; CANEDO *et al.*, 2020), and the impacts of their roles into activities as code review (BOSU; CARVER, 2014) and code refactoring (ALOMAR *et al.*, 2020, 2021).

Such extensive knowledge has primarily focused on developers' actions within software repositories. However, the quality of the developers' contributions may also vary according to their roles in the open-source project. As pointed out by Terceiro, Rios and Chavez (2010), core developers have a deeper knowledge of the software architecture; therefore, their source code changes do not add as much structural complexity as the changes made by peripheral developers do. Even more, the changes made by core developers may also remove structural complexity. Similarly, the developers' roles may influence the quality of other software artifacts.

This empirical study aims to understand the relationship between the developers' roles and test code quality. First, we investigate the survivability (insertion and removal) of test smells in open-source projects throughout the evolution of software projects. Second, we investigate whether the developers' roles influence the survivability of test smells. To accomplish such goals, we analyzed all commit history of four open-source Java projects to collect data on the insertion and removal of 16 test smells and their authorship. The yielded results indicate that 1,081 test smells were successfully removed. Among these, 122 instances (11.30%) were removed due to refactorings, while the rest of the test smells were removed due to code deletion. Of the 5,181 instances, 545 (10.39%) were removed from 366 to 2,911 days. Moreover, our analysis revealed a trend wherein peripheral developers were responsible for removing a larger number of test smells, but core developers addressed different types of test smells.

7.2 RESEARCH METHODOLOGY

We formulate two research questions (RQs) to guide our study:

RQ1 **What is the lifetime of test smells in software projects?** We aim to analyze the duration of test smells, from their introduction to their (potential) removal, to understand their survivability during software evolution.

RQ2 **Does developer experience affect the survivability of test smells?** We investigate whether developers' experience level in a project impacts the presence and removal of test smells.

7.2.1 Study Design

Our study design consists of four main steps, as Figure 7.1 shows.

Step 1. Project Selection: We selected open-source Java projects using the SEART

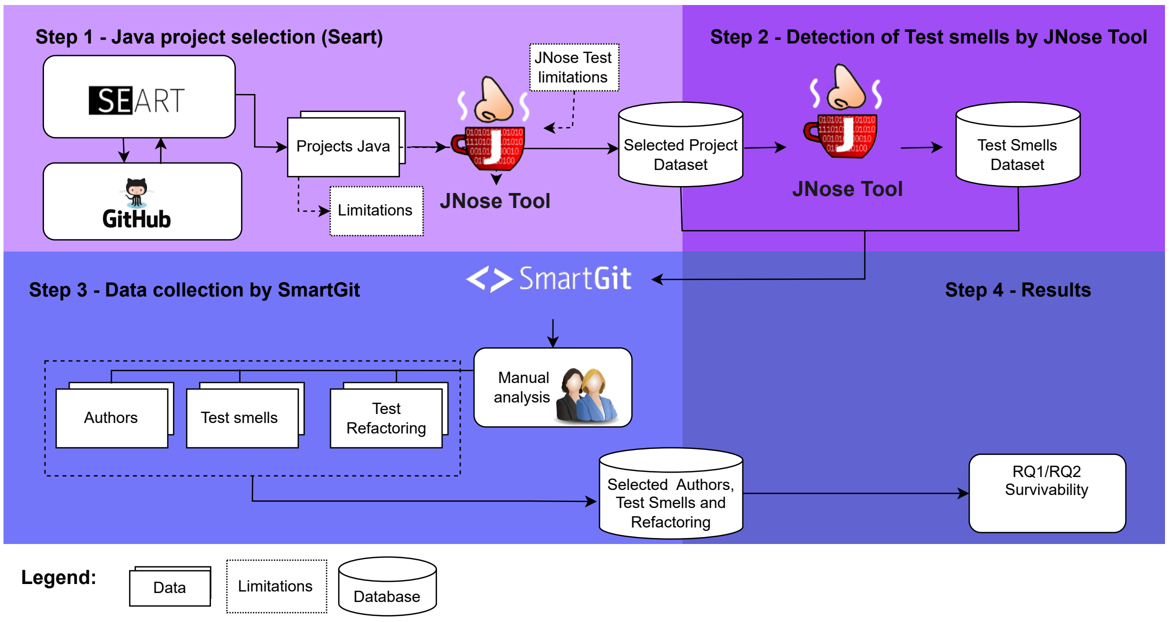


Figure 7.1: Study Design

TOOL. We applied the following criteria: (i) repositories with at least ten stars, (ii) Java-based projects, (iii) non-forked projects, (iv) mature projects that continuously receive community support, with longevity equal to or greater than five years and at least five years of releases, (v) projects containing at least ten tags (for this study, we consider the tags as a new project version), and (vi) projects containing at least ten test classes. Using stars as an evaluation criterion is based on the study by Borges and Valente (2018), which emphasizes their relevance in analyzing the evolution of software projects hosted on GitHub repositories. After retrieving 55 projects with SEART TOOL, the next step was to analyze them with the *JNose Test* tool. Only ten projects met the requirements of the *JNose Test* tool, as described in Section 7.2.3. Furthermore, all of these projects had a set of JUnit4 test cases. We randomly selected four projects from this set of selected projects.

Step 2. Detection of Test Smells: We detected test smells using the *JNose Test* tool. It generates a list of test smells in the selected projects by creating an abstract syntax tree and performing static analysis of the test code. The tool provides two *.csv* files: one for the test smells detected in the most recent commit, and another for the test smells detected in each project version.

Step 3. Data Collection: In this step, we manually collected data from the selected projects and the test smells detected in Steps 1 and 2 using the SMARTGIT tool. We used the tool to get project logs, and through the logs, we collected some metadata, such as the author of the commit, the set of commits composing a project, and the set of detected test smells. The SmartGit tool provides us with the entire history of the projects, from the creation of the first commit to the last commit performed, i.e., the complete commit history of the projects.

Step 4. Analysis of Results: In this phase, we quantitatively analyzed the data collected in Step 3 to answer the research questions. We investigate the survivability of test smells and the impact of developer experience on test smell removal. We also perform exploratory data analysis to identify patterns and trends in the data. Finally, we interpret the results and draw conclusions that address the research questions.

7.2.2 Classification of core and peripheral developers

This section presents the definitions of the developer's roles and experience. The developer's experience in the project is defined based on the number of commits performed by each developer (TERCEIRO; RIOS; CHAVEZ, 2010; CALEFATO *et al.*, 2022). We computed a threshold at the 80% percentile to categorize developers' experience in the project. Developers who have made more commits than the threshold are considered core developers, while those who have made fewer commits are considered peripheral developers. This categorization helps us better understand the level of involvement of each developer in the project.

7.2.3 Data Collection

During the data collection phase, we encountered the saturation effect, a well-known phenomenon in which the convergence rate reaches a point where no more progress can be made towards specific criteria or goals (LYU, 1996; EWASCHUK; BEYER, 2016). To mitigate this effect, we randomly selected four Java open-source projects for data collection: *apolloconfig/apollo*[1], *airbnb/lottie-android*[2], *code4craft/webmagic*[3], and *hs-web/hsweb-framework*[4]).

We utilized two, namely SmartGit and *JNose Test*, to collect data for our study. It is important to note that *JNose Test* relies on the JUnit naming convention to identify test classes. Specifically, it adds the word 'Test' as a prefix or suffix to the name of the production class. For instance, if the production class name is Job.java, then the

[1]Available at: ⟨https://github.com/apolloconfig/apollo⟩
[2]Available at ⟨https://github.com/airbnb/lottie-android⟩
[3]Available at: ⟨https://github.com/code4craft/webmagic⟩
[4]Available at: ⟨https://github.com/hs-web/hsweb-framework⟩

corresponding test class names would be JobTest.java or TestJob.java. Additionally, we only analyzed classes with the name classnametest.java that were detected by the *JNose Test*. Although we observed other class names, such as TestJob.java in the hsweb-framework project, we solely considered those identified by the *JNose Test* for our study. We also limited our analysis to files with the .java extension and excluded files with other extensions, such as TK.

Moreover, we analyzed each project module individually as they can contain test classes with the same name but different test methods and smells. For instance, the hs-web/hsweb-framework project has the hs-web/hsweb-authorization and hs-web/hsweb-core modules. Therefore, we counted the instances of test smells for each module separately.

We complemented our data collection with a manual analysis of two test smells, namely *AR* and *IgT*. We considered a test method to have an *AR* if it contains two assertions without argumentation with the same parameters (i.e., *DA*) or two numbers as parameters (i.e., *MNT*). Although *JNose Test* only counts one of them as an *AR*, we counted both if the test method had two of these test smells. Similarly, we counted each assertion without documentation as an *AR*. Furthermore, we noted all test smells in the test class during the manual analysis, even if they were ignored using the *@Ignore* tag. To enhance the study's validity, a data sample collected by one researcher was co-validated by other researchers.

When tracking commits, we observed that *JNose Test* only detects test smells present in the last commit of a tag. Therefore, we needed to track the exact location of each test smell to determine if it was inserted or removed and any modifications made to the test file containing the test smell. Additionally, we checked if consecutive tags kept the same test smells to avoid duplicate instances of test smells throughout the project's evolution. Unfortunately, we did not find any available tool to detect the insertion and removal of test smells even if we ran the tool on every commit (ALJEDAANI *et al.*, 2021; PANICHELLA *et al.*, 2022). Thus, we manually added test smells that were not detected by *JNose Test* in projects.

To analyze the survivability and lifetime of each test smell in the project, we collected several data points, including the test class name, start date of insertion, end date of removal, the first commit of the test class where the test smell was detected (i.e., the test class where the test smell was introduced), and the first commit of the test class where the test smell was removed (i.e., the test class where the test smell was fixed). To calculate the lifetime of each test smell, we used the Excel function DATEDIF (MEHTA, 2021). DATEDIF is a date function that calculates the difference between two dates in terms of days, months, and years. We used it to determine the number of days each test smell survived in the project. Additionally, we computed the average and median lifetime of each test smell and their standard deviation. These metrics provide insights into the overall resilience of the test suite and the potential impact of each test smell on the codebase's maintainability.

7.3 RESULTS

In this section, we present the yielded results, which are divided into subsections, as follows: Section 7.3.1 presents the results related to RQ1, while Section 7.3.2 presents the results related to RQ2. Data is publicly available in an online open data repository (CAMPOS *et al.*, 2023a).

7.3.1 Lifetime of test smells in software projects (RQ1)

To address RQ1, we analyzed the duration of test smells in software projects and investigated their survivability during software evolution. We collected data on the insertion and removal dates of 16 types of test smells from four software projects: *airbnb/lottie-android*, *code4craft/web-magic*, *hs-web/hsweb-framework*, and *apolloconfig/apollo*. We also considered the experience level of developers in the projects regarding removing test smells.

Figure 7.2 illustrates the distribution of removals across three distinct periods. Our analysis shows the survivability of test smells spanning from 1 to 2,911 days. These test smells were removed into three timeframes: 1-100 days, 101-365 days, and 366-2,911 days. Notably, the most substantial removals (331 instances, 30.62%) occurred during the initial 100 days of project involvement. An additional 205 instances (3.39%) were addressed from day 101 to day 365. In comparison, 545 instances (50.42%) were eliminated after a year within the system, indicating a persisting presence of 4,097 instances (79.83%) in the projects. Among the specific test smells, *RA*, *RO*, and *SE* experienced merely 1 to 2 removals throughout the project's duration, whereas *AR*, *MNT*, and *PS* stood out as the test smells with the highest removal frequency. This trend is visually summarized in Figure 7.3, which showcases the distribution of test smells by their periods of survivability and emphasizes the retention of 5,178 test smells within the projects.

For our analysis, the *apolloconfig/apollo* project removed 14 types of test smells. Test smells *EpT, ECT, DA, ST, PS,* and *RO* were removed from the analysis due to only one

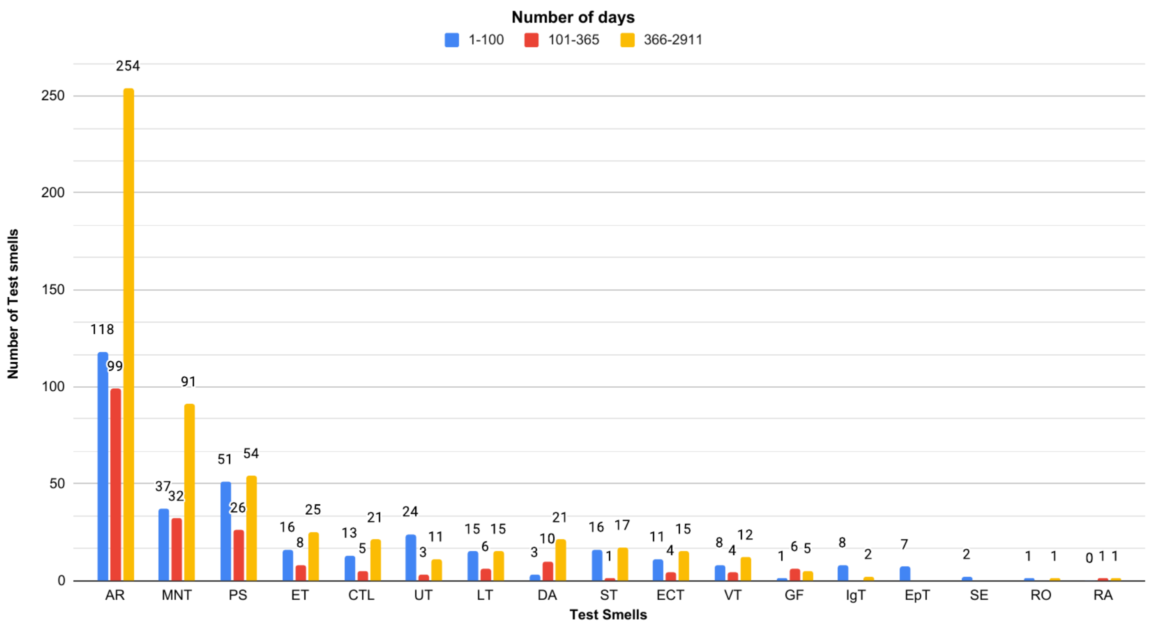


Figure 7.2: Classification of the longevity of test smells

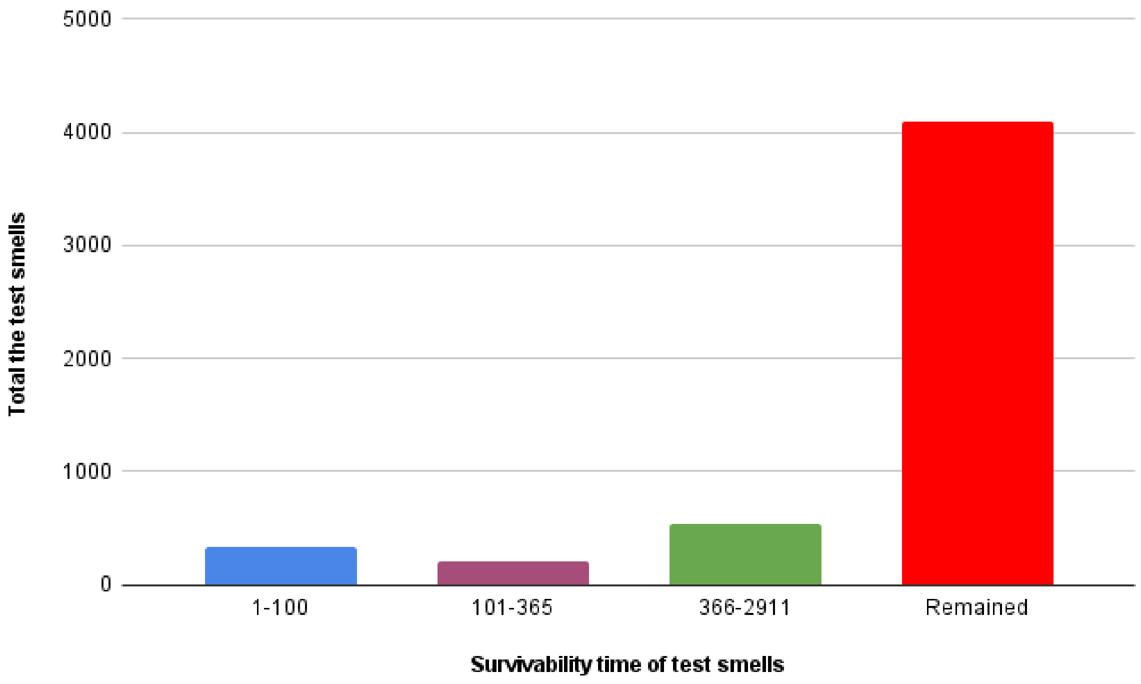


Figure 7.3: Longevity and remained of test smells

removal instance. Figure 7.4a shows that the removal of test smells *AR, MNT, GF, VT, ET*, and particularly test smell *LT* were concentrated in the third quartile, whereas only

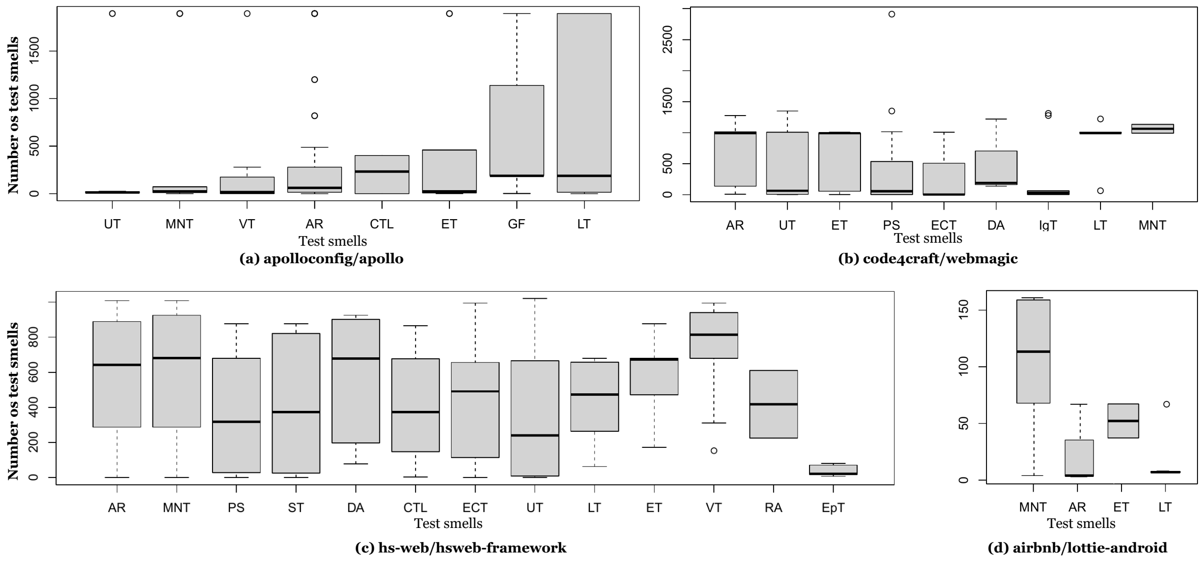


Figure 7.4: The test smells regarding removal in software projects

test smell *CTL* was concentrated in the second quartile. Test smell *UT* had a median survivability of eight days, with its removal being concentrated at the median.

In the *code4craft/webmagic* project removed 12 types of test smells, excluding *EpT* and *RO* as they had only one instance of removal. Figure 7.4b reveals that the removal of test smells *AR* and *ET* is concentrated in the second quartile, while test smells *ET, ECT, PS,* and *UT* are concentrated in the third quartile. Test smell *IgT* had ten instances of removals, but its median code survivability was only 29 days. Test smell *SE* had two instances of removals, but its median code survivability was 30.5 days. The median survivability of test smell *LT* was 991 days despite having only five instances of removals. Similarly, test smell *MNT* had a median permanence of 1,062.5 days despite being removed only twice. Test smells *VT* and *CTL* had three instances of removals each, and their median survivabilities were 55 and 1015 days, respectively.

In the *hsweb/framework* project, the 13 types of test smells were examined. However, *GF* was removed from the analysis due to having only one instance of removal. As depicted in Figure 7.4c, the distribution of the number of days of the removal of test smells *AR, DA, ET, ECT,* and *MNT* was concentrated in the second quartile. On the other hand, the removal of test smells *CTL, EPT, PS, UT,* and *ST* was concentrated in the third quartile. The removal of the test smells *LT, VT,* and *RA* was concentrated in both the second and third quartiles.

Lastly, five types of test smells were removed in the *airbnb/lottie-android* project. Test smell *ECT* was removed from the analysis because it had only one instance of removal. As seen in Figure 7.4d, the removal of test smell *AR* was concentrated in the third quartile, while test smells *ET* and *MNT* were concentrated in the second and third quartiles, respectively. Despite having seven instances of removals, test smell *LT* had a median survivability of only seven days.

In total, 16 types of test smells were removed from the analysis, excluding *RO* and *EpT*, which had only one removal instance each. When looking at the distribution of the number of days of the removal of the remaining test smells, we can observe that *AR, MNT, ECT, DA, VT*, and especially *LT* were mostly concentrated in the second quartile. On the other hand, test smells *GF, UT, CTL*, and *Ept* were mainly concentrated in the third quartile. The removal of test smells *RA* and *ST* were spread across both the second and third quartiles, as depicted in Figure 7.5.

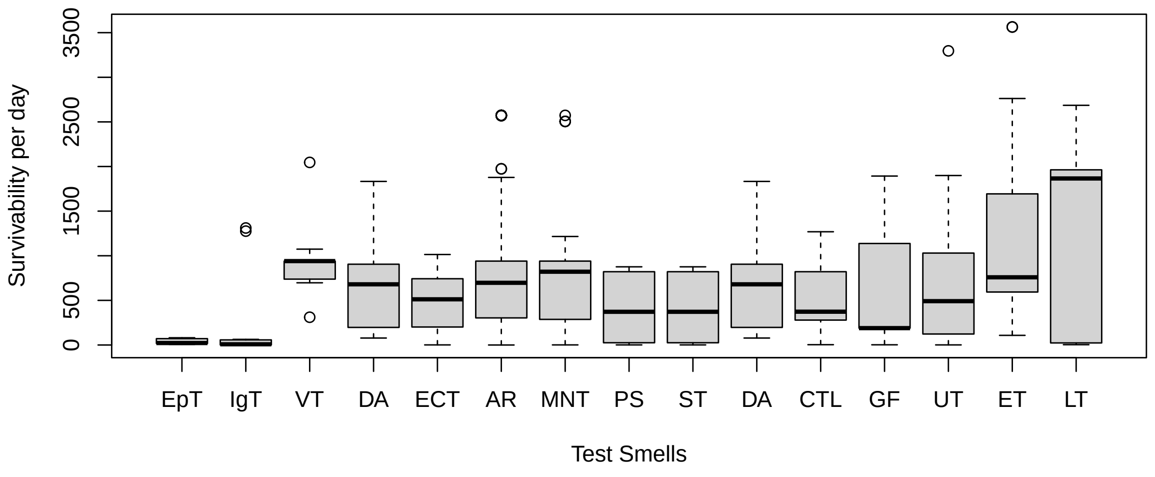


Figure 7.5: Survivability of test smells concerning the removal of test smells in the project

7.3.2 Impact of developer experience on the survivability of test smells (RQ2)

To answer RQ2, we analyze only the refactorings of test cases by removing test smells, focusing on the project history and the developer's experience during software evolution. In this analysis, we distinguish between core developers, who are responsible for creating and removing production classes and test classes from the project and are members of the core team, and peripheral developers, who are more numerous and typically perform only bug fixes (TERCEIRO; RIOS; CHAVEZ, 2010). Our analysis shows that the experience and role of developers have an impact on the survivability of test smells.

Our analysis of the *hsweb/hsweb-framework* project, which contains 17 contributors and 714 removed test smells, showed that only 41 were removed by refactoring test cases. Table 7.1 presents the tags of the project where the refactorings occurred, the type of developer who performed the refactoring, the number of refactored tests, and the types of test smells refactored in each tag. In the v3.0.0-Snapshot version of the project (rows two and three), we observe that core and peripheral developers refactored test smells, with the peripheral developer refactoring 19 test smells and the core developer refactoring nine test smells. Interestingly, we also observed that the same core developer performed refactoring activities across all tags in this project and removed three more test smells than the peripheral developer.

Table 7.1: *hsweb/hsweb-framework* projects

Version	**Developers**	**NTS**	**Test smells**
v2.2	Core	5	PT/AR/CTL
v3.0.0 -Snapshot	Core	9	PT/MNT/ECT/ST
v3.0.0 -Snapshot	Peripheral	19	AR/CTL/ECT/PT
v3.0.6	Core	2	PT
v4.0.0	Core	6	MNT/AR/UT

Legend: Number of Test Smells (NTS)

In the *apolloconfig/apollo* project, we identified 254 test smells, of which only 55 were removed through test case refactoring. Table 7.2 presents the tags where the refactoring of test cases with test smells occurred in the first column. The second column displays the developers who performed the test code refactoring. The third column shows the total number of test smells removed through refactoring, and the fourth column displays the types of test smells refactored in each tag. Interestingly, only three out of the 91 project contributors performed the refactoring, indicating a relatively low contribution from peripheral developers in test smell removal through refactoring. Two of the three developers who performed refactoring were core developers in other tags but acted as peripheral ones in the refactored tags.

In the *airbnb/lottie-android* project, only 4 out of 24 test smells were removed by refactoring the test cases, as shown in Table 7.3. This project has 108 contributors, but only one developer performed the test code refactoring. The removal of test smells was observed in just three project tags. Notably, only one developer was responsible for the refactoring, and this developer was present in all project tags as either core or peripheral. For instance, in v2.5.5, the developer acted as core, while in v2.7.0, the same developer acted as peripheral. Thus, a single developer removed 4 test smells, corresponding to 3

Table 7.2: *apolloconfig/apollo* projects

Version	**Developers**	**NTS**	**Test Smells**
v.0.4.0	Peripheral	33	AR/CTL/ECT/GF/MNT/RO/ UT/VT/ET/LT/PS
v0.5.0	Peripheral	12	MNT/VT/AR
v0.9.0	Peripheral	4	CTL
v1.5.0	Peripheral	4	LT/AR
v1.6.0	Peripheral	2	CTL

different types of test smells.

Table 7.3: *airbnb/lottie-android* projects

Version	Developers	NTS	Test smells
v2.5.5	Peripheral	1	AR
2.6.0 rc1	Core	1	ECT
v2.7.0	Core	2	MNT/AR

Legend: Number of Test Smells (NTS)

In the *code4craft/webmagic* project, out of the 89 test smells removed, 22 were eliminated by refactoring the test cases, as Table 7.4 shows. This project had a relatively larger number of refactorings than the others studied. These refactorings were performed in 10 different tags by a single developer, who played core and peripheral roles. The *code4craft/webmagic* project had 42 contributors, but only three developers were responsible for removing the test smells. This developer refactored 11 test smells and eliminated them in various project tags.

Table 7.4: *code4craft/webmagic* projects

Version	**Developers**	**NTS**	**Test smells**
v0.1.0	Core	3	SE/PS/IgT
v0.2.0	Core	3	ET/PS
v0.3.1	Core	2	AR
v0.5.2	Core	2	ECT/PS
v0.5.3	Core	1	IgT
v0.6.0	Core	1	AR
v0.7.0.alpha	Core	4	MNT/LT/DA/AR
V0.7.1	Core	4	UT/PS/IgT/EpT
v0.7.2	Core	1	IgT
v0.7.5	Peripheral	1	PS

Legend: Number of Test Smells (NTS)

In the *hsweb/hsweb-framework* project, 41 test smells were removed through refactoring, with 22 removed by the core developers and 19 removed by the peripheral developers.

This resulted in eliminating eight different types of test smells *(PS, AR, CTL, MNT, ECT, ST, UT, ET)*. Peripheral developers removed four types of test smells *(PS, AR, CTL, ECT)*, while the core developers removed all eight test smells.

Regarding the *apolloconfig/apollo* project, a total of 55 test smells were removed through test code refactoring, targeting 11 different types of test smells *(AR, CLT, ECT, GF, MNT, RO, UT, VT, ET, LT, PS)*. Among them, 41 test smells were removed by the core developers, who acted as peripherals in the specific tag where the refactoring took place.

The *airbnb/lottie-android* project had four test smells removed, specifically one ECT, two AR, and one MNT. The core developer removed three types of test smells *(ECT, AR, MNT)*, while the peripheral developer removed one type of test smell *(AR)*.

In the *code4craft/webmagic* project, 22 test smells were removed, targeting 11 different types of test smells *(SE, PS, IgT, ET, AR, MNT, LT, DA, UT, EpT, ECT)*. Among them, the peripheral developer removed only one type of test smell *(AR)*.

Our analysis underscores a consistent trend: core developers were primarily accountable for the removal of 90.47% of identified test smells, while the responsibility for the remaining 9.53% fell to peripheral developers. In the initial 100-day span, among 1,081 instances of test smells, 331 were effectively eliminated, with 79.46% tackled by core developers and 20.54% by peripheral counterparts. Shifting to days 101 to 365, core developers addressed all 205 identified test smells. Extending the timeline to 366 to 2,911 days, core developers managed the resolution of most, specifically 510 instances (93.58%) out of 545 total test smells, in contrast to peripheral developers who handled just 35 cases (6.42%). These findings mirror those of Tufano *et al.* (2016), where even after 2,911 days and considering 16 test smell types, a substantial number, 4,097 instances (79.83%), persisted without resolution.

In summary, core developers performed 46 test code refactorings, while peripheral developers removed 76 test smells through test code refactoring. Although the core developers removed fewer test smells, they eliminated 14 types of test smells *(AR, PS, CTL, MNT, ECT, ET, IgT, UT, LT, ST, DA, EpT, SE)*. On the other hand, the peripheral developers removed 11 different types of test smells *(AR, PS, CTL, MNT, ECT, ET, UT, GF, LT, VT, RO)*. Hence, core developers removed three more types of test smells than peripheral developers. The overall results are presented in Table 7.5.

We performed two Spearman correlation rank tests. The first correlation was between the developer's experience and the number of test smells inserted, for which we obtained the value of $\rho = 0.3897893$. The second correlation was between developer experience and the number of test smells removed, for which we got the value of $\rho = 0.3778612$. Both correlations show a moderate positive correlation between the two variables. The result points out that there is no statistically significant difference.

Table 7.5: All projects

Test Smells	Core	Peripheral
Assertion Roulette (AR)	10	29
Print Statement (PS)	12	13
Conditional Test Logic (CTL)	2	12
Magic Number Test (MNT)	6	6
Exception Catching Throwing (ECT)	2	3
Eager Test (ET)	2	3
Ignored Test (IgT)	4	0
Unknown Test (UT)	2	1
General Fixture (GF)	0	3
Lazy Test (LT)	1	2
Verbose Test (VT)	0	3
Sleepy Test (ST)	2	0
Resource Optimism (RO)	0	1
Duplicate Assert (DA)	1	0
Empty Test (EpT)	1	0
Sensitive Equality (SE)	1	0
Total	46	76

7.4 DISCUSSION

This section delves into and analyses the findings of each RQ discussed in this study.

7.4.1 Lifetime of test smells in software projects (RQ1)

Our study investigated the removal of test smells by analyzing the deletion or refactoring of test classes and methods during software evolution. We found that out of the 1,081 test smells we analyzed, 545 (50.42%) remained in the code for approximately seven years (2,911 days). On the other hand, 331 (30.62%) of the test smells were removed within the first 100 days, which is an average of three months to remove 331 test smells.

We also observed that the test smells *AR, MNT,* and *PS* had the highest number of removals over time, respectively. Interestingly, the *AR* test smell was the most frequently removed in the 1-100 days and 366-2,911 days, with 118 and 254 removals, respectively. However, it also had the highest survivability rate, with only 471 out of 28,836 insertions removed, leaving 2,362 (83.29%) test smells remaining in the projects. The *MNT* test smell was the second most frequently removed during software evolution, with 160 removals. In contrast, the *RA, RO,* and *SE* test smells were rarely removed, with only 1 to 2 insertions/removals in the projects.

It is worth noticing that these findings regarding the lifetime of test smells are harmful to the test code and can hinder the understanding of the test code, maintainability, and lead to technical debt.

7.4.2 Impact of developer experience on the survivability of test smells (RQ2)

To address this research question, we examined 16 different test smells (*AR, CTL, ECT, MNT, PS, ST, UT, ET, GF, LT, RO, VT, DA, EpT, IgT, SE*) across four open-source software projects. The *code4craft/web-magic* project had 11 test smells, while the *hsweb/hsweb-framework*, *apolloconfig/apollo*, and *airbnb/lottie-android* projects had 7, 11, and 3 test smells, respectively. The *AR* test smell was present in all projects, and six test smells were present in three projects.

Of the 1,081 test smells removed during software evolution, 122 (11.28%) were removed through test case refactorings. Core developers were responsible for removing 37.7% of these refactored test smells, while peripheral developers refactored the remaining 62.3%. The core developers refactored 13 different test smells, and the peripheral developers refactored 11 ones.

The findings suggest that peripheral developers are the primary contributors to test smell removal. Still, core developers play a crucial role in test case maintenance and restructuring to improve the co-evolution of production classes.

7.5 THREATS TO VALIDITY

Internal validity. The present study only considered open-source, Java, and Git-based repositories. While we selected projects that vary in size, contributors, number of commits, and refactoring strategies to mitigate this threat, the findings may not generalize to projects developed in other programming languages, version control systems, or development environments. Additionally, our data collection involved manual processes, which may introduce errors or biases. To address this concern, multiple researchers collected and verified the data to increase its reliability. In this study, we focus on the developer's experience in the project through the number of commits. There are other study targets, such as developer activities and organization policy. However, these studies are outside our scope because we can identify the time it takes to remove test smells.

Nevertheless, we can not know the policies, personal motivation, or other explanations for developers removing test smells.

Construct validity. Several factors can influence our study's results. First, the interpretation of data can be subjective, particularly when identifying test smells' presence and reasons for refactoring. Second, using the Proportion of Ownership technique to measure developer experience may not capture developers' skills or expertise. Lastly, our use of the JNose tool to detect test smells is limited to classes following the JUnit terminology, and thus, we may have missed some test classes with test smells. However, we manually validated the tool's results and took steps to ensure data accuracy and completeness.

External Validity. The results of our study are based on a sample size of four open-source Java projects, which may limit the generalizability of the findings to other software projects. For example, some projects find only one developer who has removed the test smells. These projects also vary in size, number of test classes, and number of commits, which may affect the results. Additionally, our analysis focuses on test smells, which are specific types of code smells found in test code. It may not generalize to other types of code smells or code quality issues. While we provide a detailed methodology of our study, future replications may benefit from examining other software projects or using different tools to detect test smells.

Conclusion validity. Our investigation into the frequency of developers addressing test smells relied on an automated tool identifying test smells across varying levels of code granularity. For instance, AR detection occurred at the line level, UT detection at the method level, and IgT detection at the class level. It is worth noting that the granularity of test smell detection could potentially impact the outcomes. Furthermore, we conducted statistical tests to examine the relationship between developers' roles and the insertion or removal of test smells. However, it is important to acknowledge that such analyses might be susceptible to various confounding variables, which were not accounted for in our study.

7.6 CONCLUSION

This chapter presented the empirical study carried out to examine test smell survivability and the interplay between developer experience and refactorings in four open-source GitHub projects. Our analysis, involving 16 test smells, 183 tags, and 256 contributors, yielded significant insights. We found that only 122 (11.30%) of the 1,081 test smells were eliminated through test case refactoring, suggesting that refactoring often doesn't target test smell removal. Furthermore, 331 (6.39%) test smells were removed in the initial 100 days, 205 (3.39%) from days 101 to 365, and 545 (10.39%) from days 366 to 2,911, revealing a notable 4,097 (79.83%) permanence within projects. Peripheral developers handled more test smell refactoring, while core developers addressed a wider array, attributed to their diverse role in enhancing test code quality.

Our findings emphasize the mismatch between test case refactoring and test smell removal, highlighting the persistence of test smells. They also highlight the distinct roles of different developer types and the direct impact of changes in production classes on test code.

Chapter

8

TEST SMELLS KNOWLEDGE IN STACK EXCHANGE COMMUNITIES

This Chapter presents the empirical study we carried out to answer **RQ6** and **RQ7** of this thesis. This empirical study aims to understand how developers discuss test smells and their refactorings on Stack Exchange, the leading collaboration network for software development. This study contributes a synthesis of community discussions about test smells and the main challenges, test design issues, test code refactoring, and post-refactoring test behavior (MARTINS *et al.*, 2023b).

We structured this chapter as follows: Section 8.1 introduces the empirical study. Section 8.2 presents the research methodology. Section 8.3 presents the results. Section 8.4 discusses the results. Section 8.5 discusses the threats to validity. Finally, Section 8.6 concludes the Chapter.

8.1 INTRODUCTION

The Software Engineering research community has dedicated efforts to understanding anti-patterns and proposing solutions to assist developers in refactoring the code to cope with design issues (PALOMBA *et al.*, 2014; GAROUSI; KÜÇÜK, 2018; HOZANO *et al.*, 2018; SPADINI *et al.*, 2020). However, the evolution of programming languages and frameworks creates new anti-patterns that require novel refactorings to fix them, making it challenging for the community to keep up with the actual problems developers face in practice (GAROUSI; PETERSEN; OZKAN, 2016). For this reason, developers often post questions on Q&A platforms seeking help to solve a problem with their code. In a recent study, Peruma *et al.* (2022) conducted quantitative and qualitative experiments to understand how developers discuss refactoring in a collaborative online discussion forum. The authors aimed to unveil the most explored and discussed topics concerning software refactoring.

By analyzing the literature on test smells (GAROUSI; KÜÇÜK, 2018; SPADINI *et al.*, 2020; KIM, 2020), we may observe little evidence of the challenges developers face during the automated test code development and maintenance and the commonly discussed issues, mainly unit testing. Furthermore, little is known about commonly used corrective strategies and strategies that ensure the preservation of test behavior after test code refactoring.

We conducted an empirical study to understand how the developers discuss test smells and test refactorings in the Stack Exchange, the leading software development collaboration network. By analyzing data from Stack Exchange, we could contribute with a synthesis of the community discussions about test smells, mainly regarding the key challenges, test design issues, test code refactoring, and post-refactoring test behavior.

8.2 RESEARCH METHODOLOGY

This study addressed the following Research Questions (RQ):

RQ$_1$ What challenges do developers report for handling problems in the test code? This RQ studies developers' main difficulties in refactoring test code by grouping the questions developers ask into *why-how-what* categories.

RQ$_2$ What test smells do developers most actively discuss? This RQ studies which test smells developers consider relevant to refactor and classifies them following Garousi et al.'s catalog (GAROUSI; KÜÇÜK, 2018).

RQ$_3$ What preventive and corrective actions do developers suggest to handle test smells in the test code? This RQ leverages the actions developers suggest to prevent test smell insertion and the refactoring operations from fixing test smells.

RQ$_4$ Do developers discuss how to keep the test behavior after test code refactorings? This RQ investigates whether and how developers care about test behavior during refactoring.

Figure 8.1 shows the study design, which encompasses three main steps: (A) Identification of discussions, (B) Classification of discussions, and (C) Data analysis.

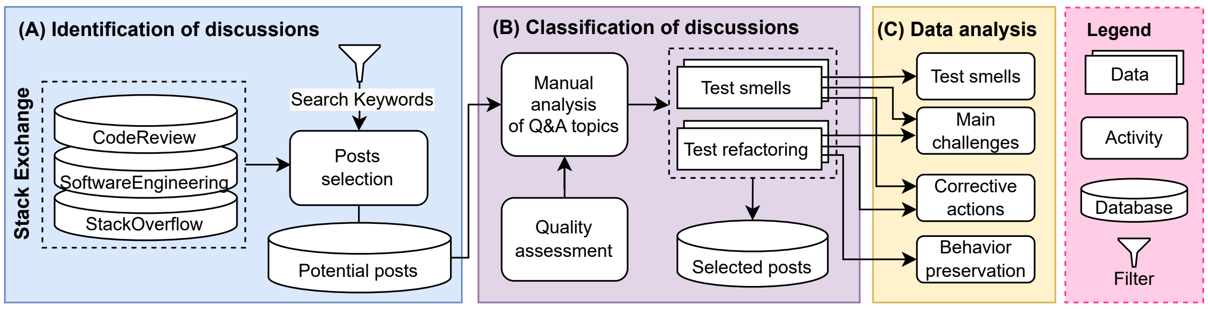


Figure 8.1: Study Design

8.2.1 Identification of discussions

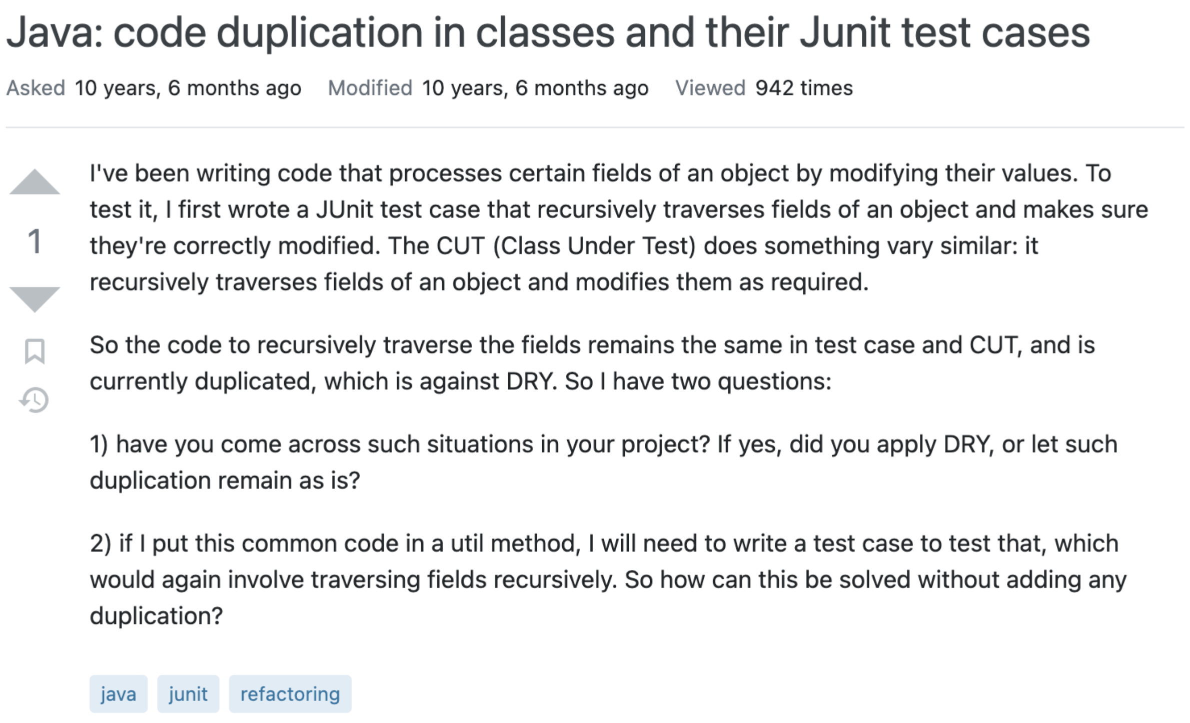
According to Tahir *et al.* (2020), developers often gather in Q&A online forums to look up how others complete similar tasks and cope with recurring issues, including how to get rid of anti-patterns. For example, *Stack Exchange* is a collaborative discussion forum network offering insightful resources on development issues (TAHIR *et al.*, 2020). Posnett *et al.* (2012) mentioned participating in and sustaining learning communities is a durable and valuable aspect of professional life.

In this study, we used the Internet Archive (IA)[1] to retrieve discussions on test code problems developers report and the corrective actions they suggest. IA keeps data dumps of discussions on the Stack Exchange network over time. We selected the following Stack Exchange sites:

- **Code Review[2]:** a site for peer programmer code reviews. It allows programmers to ask questions about specific code snippets and receive feedback from others;
- **Software Engineering[3]:** a site for developers and scholars interested in asking general questions on the systems development life cycle (TAHIR *et al.*, 2020). The site is adequate for opinion-based questions;
- **Stack Overflow[4]:** a site covering technical and general discussions on problems unique to software development (OPENJA; ADAMS; KHOMH, 2020; BHASIN; MURRAY; STOREY, 2021). The discussions commonly focus on specific programming problems or tools.

[1]Available at ⟨https://archive.org/download/stackexchange⟩
[2]Available at ⟨https://codereview.stackexchange.com/⟩
[3]Available at ⟨https://softwareengineering.stackexchange.com/⟩
[4]Available at ⟨https://stackoverflow.com/⟩

The discussions on the Stack Exchange sites associate one question post with one or more answer posts given by different users. A question post consists of a title, body, and tags. Figure 8.2 shows a sample StackOverflow question post (STACKOVERFLOW, 2012). It comes with three tags: `Java`, `JUnit`, and `refactoring`. In the post, the author asks if the practice she/he adopted in the `JUnit` test case consists of duplication and how she/he writes the test without duplication. Figure 8.3 shows a response to the question post from Figure 8.2. It reveals that the practice of the question refers to a test anti-pattern called `Ugly Mirror`. As a potential solution, the test code simplification (avoiding conditional structures and using assertions instead) and the test object creation to run the tests manually for complex data structures. The post's author rated the answer as accepted, and the answer got four votes.


Java: code duplication in classes and their Junit test cases

Asked 10 years, 6 months ago Modified 10 years, 6 months ago Viewed 942 times

1

I've been writing code that processes certain fields of an object by modifying their values. To test it, I first wrote a JUnit test case that recursively traverses fields of an object and makes sure they're correctly modified. The CUT (Class Under Test) does something vary similar: it recursively traverses fields of an object and modifies them as required.

So the code to recursively traverse the fields remains the same in test case and CUT, and is currently duplicated, which is against DRY. So I have two questions:

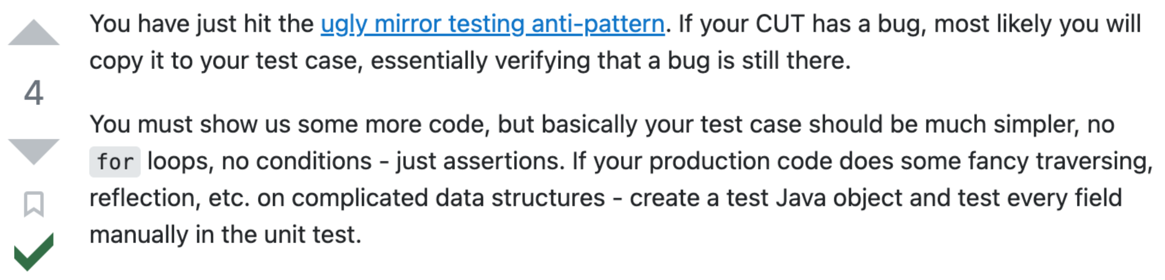
1) have you come across such situations in your project? If yes, did you apply DRY, or let such duplication remain as is?

2) if I put this common code in a util method, I will need to write a test case to test that, which would again involve traversing fields recursively. So how can this be solved without adding any duplication?

java junit refactoring

Figure 8.2: Sample question post extracted from StackOverflow.

We defined and applied a search string on the tags in each question post to select the discussions. It aimed at filtering out the discussion content based on selected tags (Figure 8.1 - *Posts selection*). Table 8.1 shows four groups of tags composing our search string. The *TEST* group filters the discussions about test codes by specifying the types of tests, testing frameworks, and language constructs used in test codes. The *DESIGN* group filters the discussions containing smell-related problems concerning test code (GAROUSI; KÜÇÜK, 2018). The *REFACTORING* group filters topics with discussions on how to


4

You have just hit the ugly mirror testing anti-pattern. If your CUT has a bug, most likely you will copy it to your test case, essentially verifying that a bug is still there.

You must show us some more code, but basically your test case should be much simpler, no `for` loops, no conditions - just assertions. If your production code does some fancy traversing, reflection, etc. on complicated data structures - create a test Java object and test every field manually in the unit test.

Figure 8.3: Accepted/useful answer from a StackOverflow post.

Table 8.1: Groups of tags composing the search string.

ID	Group	Description	Tags	Criteria
1	TEST	Words related to testing, test type, or test structure	'unit-testing,' 'testing,' 'test-scenarios,' 'unit-test-data,' 'junit,' 'junit4,' 'junit5,' 'automated-tests,' 'test-automation,' 'tests,' 'testcase,' 'assertions,' 'assert,' 'assertion,' 'annotations'	Inclusion
2	DESIGN	Words related to programming practices and design	'code-smell,' 'code-smells,' 'anti-patterns,' 'programming-practices,' 'naming,' 'naming-conventions,' 'naming-standards,' 'coding-standards,' 'coding-style,' 'code-formatting,' 'format,' 'formatting,' 'bad-code,' 'technical-debt'	Inclusion
3	REFACTORING	Words related to refactoring	'refactoring,' 'test-refactoring,' 'automated-refactoring'	Inclusion
4	LANGUAGE	Words related to programming languages	'c++,' 'c#,' 'javascript,' 'vb6', 'python,' 'python-3', 'go,' 'c,' '.net,' 'php,' 'sql,' 'ruby'	Exclusion

refactor test code. Considering that the tagged topics rarely use the words of the *REFACTORING* group, we used a logical disjunction with the *REFACTORING* and *DESIGN* groups to create a less strict filter. Similarly, many topics are untagged with programming languages. Therefore, the group *LANGUAGE* removes those topics tagged with programming languages other than `Java`.

Although some studies automatically classify thousands of Stack Exchange topics (PERUMA *et al.*, 2022; TAHIR *et al.*, 2020; CHOI *et al.*, 2015), we fine-tuned our search string to reduce the number of non-relevant topics on test smells and test refactorings (GOMES *et al.*, 2022; SANTOS *et al.*, 2023). We applied the search string "***TEST AND (DESIGN OR REFACTORING) NAND LANGUAGE***" on the data dumps of three StackExange sites from September 15th, 2008 to December 6th, 2022. As a result, we retrieved 303 potential posts.

Table 8.2: Number or retrieved discussions.

Site	**# Total posts**	**# Potential posts**	**# Selected posts**
SoftwareEngineering	237,548	153	40
CodeReview	196,301	2	2
StackOverflow	+8million	158	59

8.2.2 Classification of discussions

We manually analyzed the discussions to select the ones related to test smells and the refactorings to solve them (Figure 8.1 - *Manual analysis*). To align the analysis criteria, three coders performed a peer analysis on the CodeReview and SoftwareEngineering question topics. The coders read 155 question posts and applied the following inclusion (*IC*) and exclusion criteria (*EC*): *(IC_1)* question topics describing a problem in the test code related to bad design or implementation choices, *(EC_1)* question topics about the need for testing and how to create test code, and *(EC_2)* question topics without answers. The three researchers carried out a manual analysis. We calculated the Kappa statistics (COHEN, 1960) to assess the reliability of the manual classification (Figure 8.1 - *Quality assessment*) and reached a substantial agreement level of 0.61. Three researchers analyzed the 101 discussions. Disagreements were resolved on the basis of argument and research into the topic. If all agreed, the topic was accepted.

After, independent coders analyzed the StackOverflow question posts and accepted 59 pots (Figure 8.1 - *Selected posts*, Table 8.2 - *#Selected posts*). Next, we analyzed 101 selected posts to extract the data items listed in Table 8.3. The data extraction followed the same steps described in this section. We performed the data extraction on the CodeReview and SoftwareEngineering in peers. The coders discussed divergences in data extraction in daily meetings and performed individual data extraction on StackOverflow.

8.2.3 Data Analysis

To answer RQ$_1$, we analyzed the question topics in two steps. First, we applied a Thematic Content Analysis (TCA) (BRAUN; CLARKE, 2006) to classify the types of questions asked by developers into *why-how-what* questions (Golden-circle theory) (OPENJA; ADAMS; KHOMH, 2020): (1) *why* is a type of question that seeks to understand the reason or the cause of a problem, (2) *how* is the type of question that seeks approaches or better ways to achieve a result, and (3) *what* is the type of question to get the infor-

Table 8.3: Data items extracted from discussions.

#	**Data Item**	**Description**	**RQ**
D1	Challenges for refactoring test smells	Challenges that developers face for refactoring the test code to fix test smells	RQ_1
D2	Description of test smells	Descriptions of test smells by developers based on their understanding	RQ_2
D3	Cause of test smells	Causes that lead to test smells	RQ_2
D4	Actions for preventing test smells and refactoring the test code	Actions suggested by developers to prevent the insertion of test smells and refactor the test code to fix test smells	RQ_3
D5	Tools for refactoring test smells	Tools and sources that support developers fixing test smells	RQ_3
D6	Strategies to keep the test code behavior	Strategies to verify whether the test code behavior is kept the same after the test code refactorings	RQ_4

mation related to the problem. Then, we performed an open coding (STOL; RALPH; FITZGERALD, 2016) to interpret the question topics and extract the main challenges developers face when applying refactorings to fix problems in the test code.

To answer RQ_2, we applied a TCA to classify the discussions into (1) *specific discussion* about a specific test smell and (2) *general discussion* that does not explicitly ask about a test smell. After, we interpreted the description of the test smell and classified it according to Garousi et al.'s catalog (GAROUSI; KÜÇÜK, 2018).

To answer RQ_3, we applied a TCA in the top answers (top-rated answer, accepted answer, or unique answer) to classify their actions into (TAHIR *et al.*, 2020): (1) *Fix* to recommend test code refactorings for fixing problems in the test code, (2) *Capture* to explain the test problems but does not recommend test code refactoring to solve the problems, (3) *Ignore* to recommend ignoring taking any action to fix the test code, and (4) *Explain* why something is considered a test problem. In addition, we extracted the tools and applied open coding to list the test code refactorings suggested in the answers.

To answer RQ_4, we analyzed whether the developers asked for strategies to verify the test code behavior after performing test code refactorings. We listed the strategies and tools suggested in the answers. Data is publicly available in an online open data repository (DATA..., 2022).

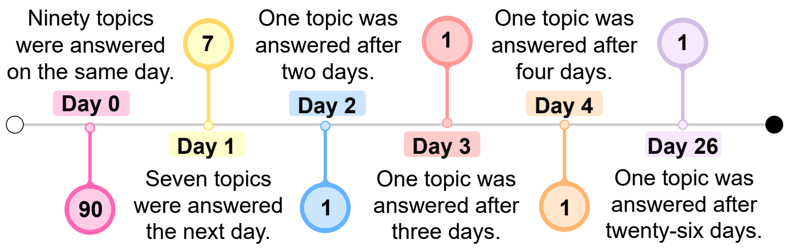

Ninety topics were answered on the same day.
Day 0
90
7
Day 1
Seven topics were answered the next day.
One topic was answered after two days.
Day 2
1
1
Day 3
One topic was answered after three days.
One topic was answered after four days.
Day 4
1
1
Day 26
One topic was answered after twenty-six days.

Figure 8.4: Time between asking a question and receiving a first answer.

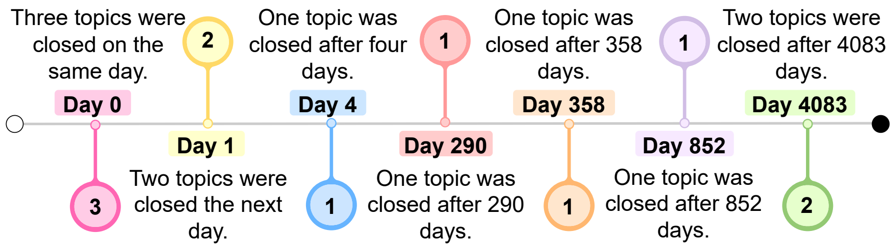

Three topics were closed on the same day.
Day 0
3
2
Day 1
Two topics were closed the next day.
One topic was closed after four days.
Day 4
1
1
Day 290
One topic was closed after 290 days.
One topic was closed after 358 days.
Day 358
1
1
Day 852
One topic was closed after 852 days.
Two topics were closed after 4083 days.
Day 4083
2

Figure 8.5: Time between asking a question and closing the discussion.

8.3 RESULTS

8.3.1 Discussions characterization

We characterized 101 discussion topics on test smells and their solutions found on the Stack Exchange network. There is nearly no repetition of users asking questions. For those users who asked more than one question, the author rephrased the question in a more specific way or on a different site. We collected 111 top answers, of which we accepted 61 answers and 38 answers were top-rated. We found another 12 answers, and although not accepted, we deemed them relevant for our investigation. Most users answered only one question, and only seven answered more than one. It is worth noting four out of these seven users are top-ten users when considering users' reputation score, number of questions, and number of answers. Next, we collected the time between asking a question and receiving the first answer (Figure 8.4) and the time to close the discussion topic (Figure 8.5). We observed that 90 out of 101 questions received an answer on the same day, and seven received an answer the next day. The remainder received an answer from the second to the twenty-sixth day. Regarding the time to close, only three topics were closed the same day they were posted, and the other 11 were closed within a year. Curiously, some took 4,083 days to close, and many never closed.

8.3.2 Trends and Challenges (RQ1)

In RQ$_1$, we aimed to understand the trends and challenges around developers' discussions on test code refactoring concepts and activities. We classified the discussions into 30 categories representing the questions asked by developers while evolving the test code. Figure 8.6 presents the number of discussions in each category and their relationship with the *why-how-what* questions.

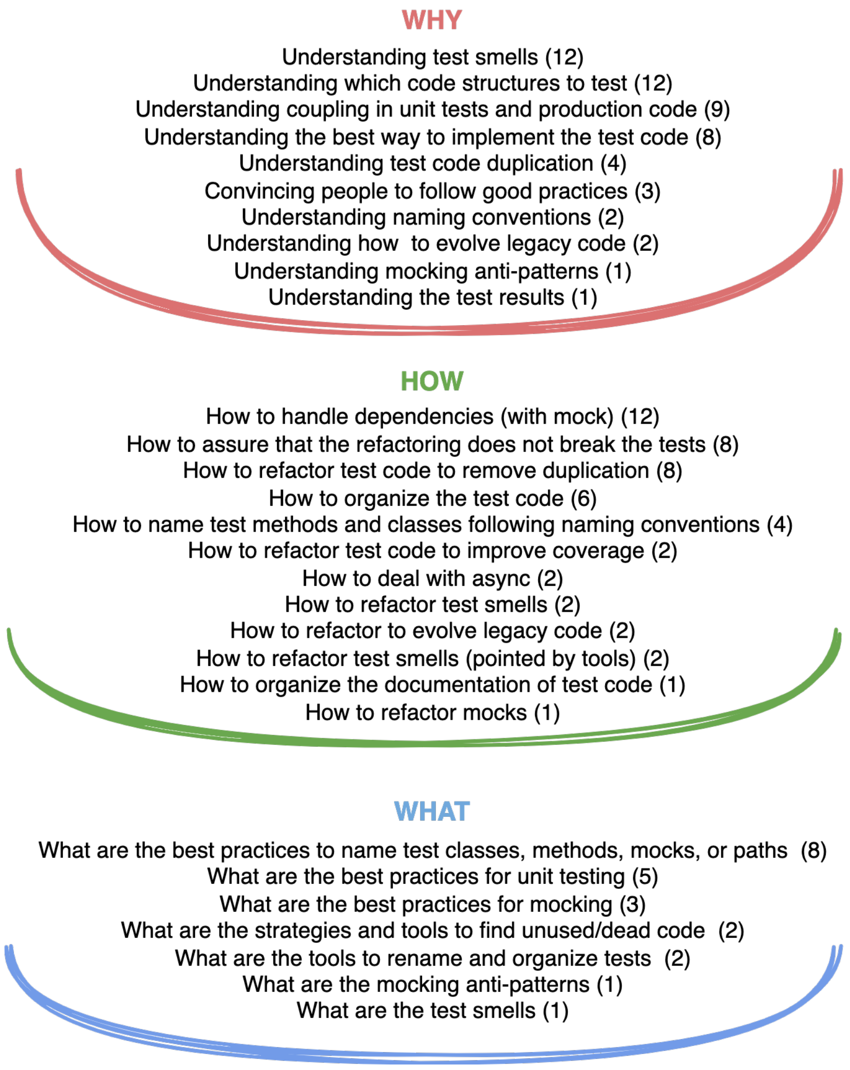

WHY
Understanding test smells (12)
Understanding which code structures to test (12)
Understanding coupling in unit tests and production code (9)
Understanding the best way to implement the test code (8)
Understanding test code duplication (4)
Convincing people to follow good practices (3)
Understanding naming conventions (2)
Understanding how to evolve legacy code (2)
Understanding mocking anti-patterns (1)
Understanding the test results (1)
HOW
How to handle dependencies (with mock) (12)
How to assure that the refactoring does not break the tests (8)
How to refactor test code to remove duplication (8)
How to organize the test code (6)
How to name test methods and classes following naming conventions (4)
How to refactor test code to improve coverage (2)
How to deal with async (2)
How to refactor test smells (2)
How to refactor to evolve legacy code (2)
How to refactor test smells (pointed by tools) (2)
How to organize the documentation of test code (1)
How to refactor mocks (1)
WHAT
What are the best practices to name test classes, methods, mocks, or paths (8)
What are the best practices for unit testing (5)
What are the best practices for mocking (3)
What are the strategies and tools to find unused/dead code (2)
What are the tools to rename and organize tests (2)
What are the mocking anti-patterns (1)
What are the test smells (1)

Figure 8.6: Classification of types of questions and challenges

In the *why* questions, we classified the discussions into ten categories to understand the test code problems and bring insights into the decision-making of whether to refactor the test code for fixing such problems. The *Understanding which code structures to test*, *Understanding coupling in unit tests and production code*, and *Understanding how* *to evolve legacy code* categories refer to problems with origin in the production code. The *Convincing people to follow good practices* category relates to problems faced by newcomers joining a team that does not follow good practices for testing the code due to organizational decisions or team conventions. The *Understanding test smells*, *Understanding the best way to implement the test code*, *Understanding test code duplication*, *Understanding naming conventions*, *Understanding mocking anti-patterns*, and *Understanding the test results* categories refer to problems related to the comprehension of the testing frameworks constructs, the importance of following good practices and conventions for developing test code, and the problems caused by the wrong usage of such constructs, practices, and conventions.

We could sort most topics into *Understanding which code structures to test* and *Understanding of test smells* categories. In the former, the discussions usually present a code excerpt asking whether the developers should test a structure of the production code (e.g., overloaded/private/public methods). In the latter, the discussions often refer to whether a particular test code is either smelly or not and why someone should be concerned about the impacts of test smells.

For example, we classified the following question in the *Understanding test smells* category. That category shows a barrier developers face in learning specific constructs for creating test cases. Although they perceive the code as smelly, they need to understand the pros/cons of improving the test code.

> (...) I always tell myself as long as the "real" code is "good," that's all that matters. Plus, unit testing usually requires various "smelly hacks" like stubbing functions. How concerned should I be over poorly designed ("smelly") unit tests? (STACKOVERFLOW, 2011)

In the *how* questions, we classified the discussions into twelve categories. These encompass strategies and techniques to handle problems reported in the *why* questions through other developers' experience. The *How to handle dependencies (with mock)* and *How to refactor mocks* categories deal with the test code evolution to use mocks and stubs for testing external dependencies with APIs, databases, web services, and files. The *How to refactor test code to remove duplication* category discusses how to handle duplication of setup methods, annotations, and test cases covering overloaded production methods. The *How to deal with async* category discusses testing framework constructs to avoid non-deterministic tests given the asynchronicity of the production code. The *How to refactor test code to improve coverage* and *How to refactor to evolve legacy code* categories involve understanding the production code to co-evolve the production and test codes, improving test coverage. The *How to organize the test code*, *How to name test methods* *and classes following naming conventions*, and *How to organize the documentation of test code* categories seek to standardize the naming, documentation, and organization of test codes. The *How to refactor test smells* and *How to refactor test smells (pointed by tools)* categories discuss strategies to fix test smells identified by developers or tools (e.g., SonarQube). Lastly, the *How to assure that the refactoring does not break the tests* category presents discussions on techniques to verify whether the behavior of the test code keeps the same after test code refactorings.

For example, we classified the following question into two categories *Understanding which code structures to test* and *How to handle dependencies (with mock)*. The question shows an example of a helper method containing some external dependency. The first category aims to understand whether the developer should test the helper method, and the second asks for strategies to implement the test code handling the dependencies of the helper method.

> So I have a helper method [..] for which I can call to grab a particular object rather than to remember which dependencies I need to hook up to get the object I require.
> My first question here is: should methods like these be tested? The only reason I can think of to test these methods would be to ensure that the correct dependencies are used and set up correctly. If the answer to the first question is yes, my second is: how? (STACKOVERFLOW, 2009a)

In the *what* questions, we classified the discussions into seven categories. The *What are the best practices to name test classes, methods, mocks, or paths*, *What are the best practices for unit testing*, and *What are the best practices for mocking* categories ask for coding standards, patterns, and guidelines for constructs in the test code. The *What are the strategies and tools to find unused/dead code* and *What are the tools to rename and organize tests* categories present tools for refactoring the test code to match the naming and structure of the production classes and improve the test code readability by removing unused code. The *What are the mocking anti-patterns* and *What are the test smells* categories ask for the anti-patterns, and test smells definitions and catalogs that can occur in the test code.

For example, we classified the following question in the *What are the good practices for unit tests* category. With this question, the developer received guidance on which patterns to use while developing the test code, e.g., the *Arrange, Act, Assert* (AAA) pattern for arranging and formatting the code.

> Usually, when talking about coding standards, we refer to the code of the program itself, but

what about the unit tests? Are there certain coding standards guidelines that are unique to unit tests? (STACKOVERFLOW, 2010)

Conversely, we classified the following question (STACKOVERFLOW, 2008) in the *What are the test smells* category. The answers to this question provided more than 31 test code anti-patterns.

There must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea [...] Vote for the TDD anti-pattern that you have seen "in the wild" one time too many. (STACKOVERFLOW, 2008)

Finding 1: Developers are interested in the practical experience of other developers to understand test smells and decide whether and how to refactor the test code to fix them.

8.3.3 Test code problems (RQ2)

In RQ_2, we investigated the test smells discussed by developers in the Stack Exchange network. We followed Garousi et al.'s catalog (GAROUSI; KÜÇÜK, 2018) to classify the posts. Although most question topics did not explicitly ask about a test smell, we could establish a link between the description of the test code problem and the test smell categorization. Therefore, we labeled the discussions without explicit test smells as **General Discussion (GD)** and the discussions with explicit test smell as **Specific Discussion (SD)** on test smells. We analyzed 36 SD with 46 test smells and 64 GD with 125 test smells.

Figure 8.7 presents the categorization of discussions into test smells. We found test smells composing seven of the eight top categories proposed by Garousi et al. (GAROUSI; KÜÇÜK, 2018). The *Code related* category refers to test smells related to the 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. The *Dependencies* category refers to test smells related to dependencies within the test code or with external resources. The *In association with production code* category refers to test smells related to coupling and dependencies between test and production code, making the tests hard to evolve. The *Mock and stub related* category refers to test smells related to misusing mock objects and mocking verification. The *Issues in test steps* category refers to occurring test smells in specific language constructs such as assertions and setup methods. The *Test execution/behavior* category refers to test smells that can lead to unexpected results as non-determinism. The *Test semantic/logic* category refers to test smells related to test logic and several responsibilities per test. The only category we did not find was the *Design related* category, which mainly presents test smells related to page-object patterns in Selenium tests.

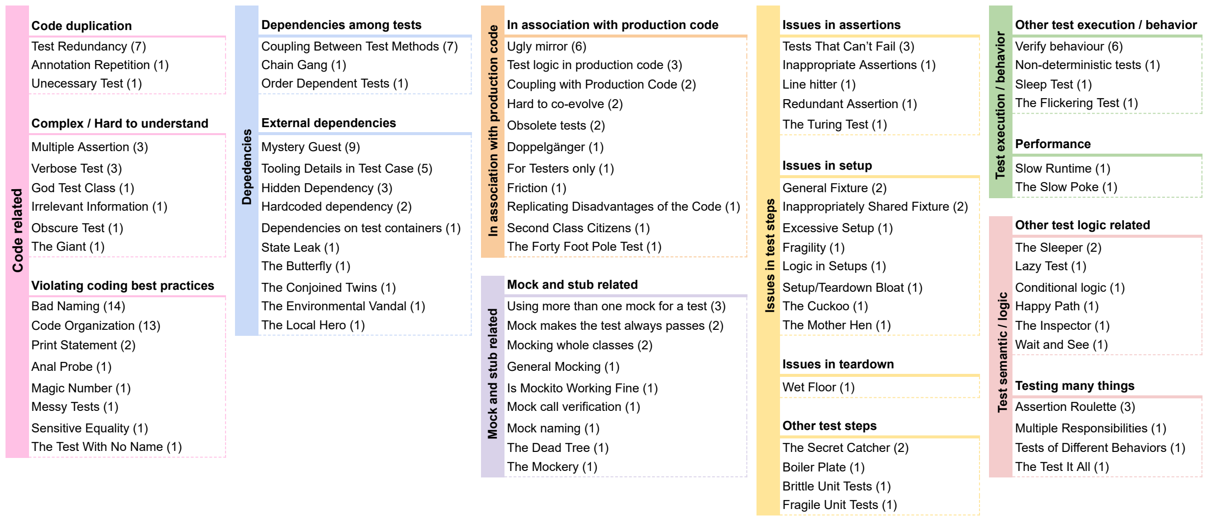


Figure 8.7: Classification of GD and SD on test smells according to Garousi's catalog of test smells (GAROUSI; KÜÇÜK, 2018).

The seven categories from Figure 8.7 group 54 test smells. The *Issues in test steps* category is the most diverse regarding test smells, comprising 18 test smells. Although the category is diverse, most of its test smells were discussed only once. Conversely, the *Code related* category is diverse and comprises two of the most recurrent test smells in the discussions. The *Bad Naming* test smell was the most frequent, with 14 occurrences. It describes the non-compliance with naming conventions for test code structures such as variables, methods, and classes. The *Code Organization* test smell was the second most frequent, with 13 occurrences. It describes the non-compliance with test code organization as following the same production and test classes package hierarchy.

In addition, the *Code related* category has the most test smells gathered from specific topics. The developers explicitly discussed ten out of 17 test smells of this category. Differently, we gathered the most test smells of other categories from the GD. While the *In association with production code*, *Mock and stub related* only contain GD, the *Dependencies* and *Issues in test steps* categories have three SD each, and the *Test execution/behavior* and *Test semantic/logic* categories have one SD each. It can indicate that developers face problems in different categories of test smells. Still, they know more about naming the test smells in the *Code related* category.

Finding 2: Developers usually ask whether something is a test smell or an anti-pattern, rather than referring to a particular one.

8.3.4 Test code refactorings (RQ3)

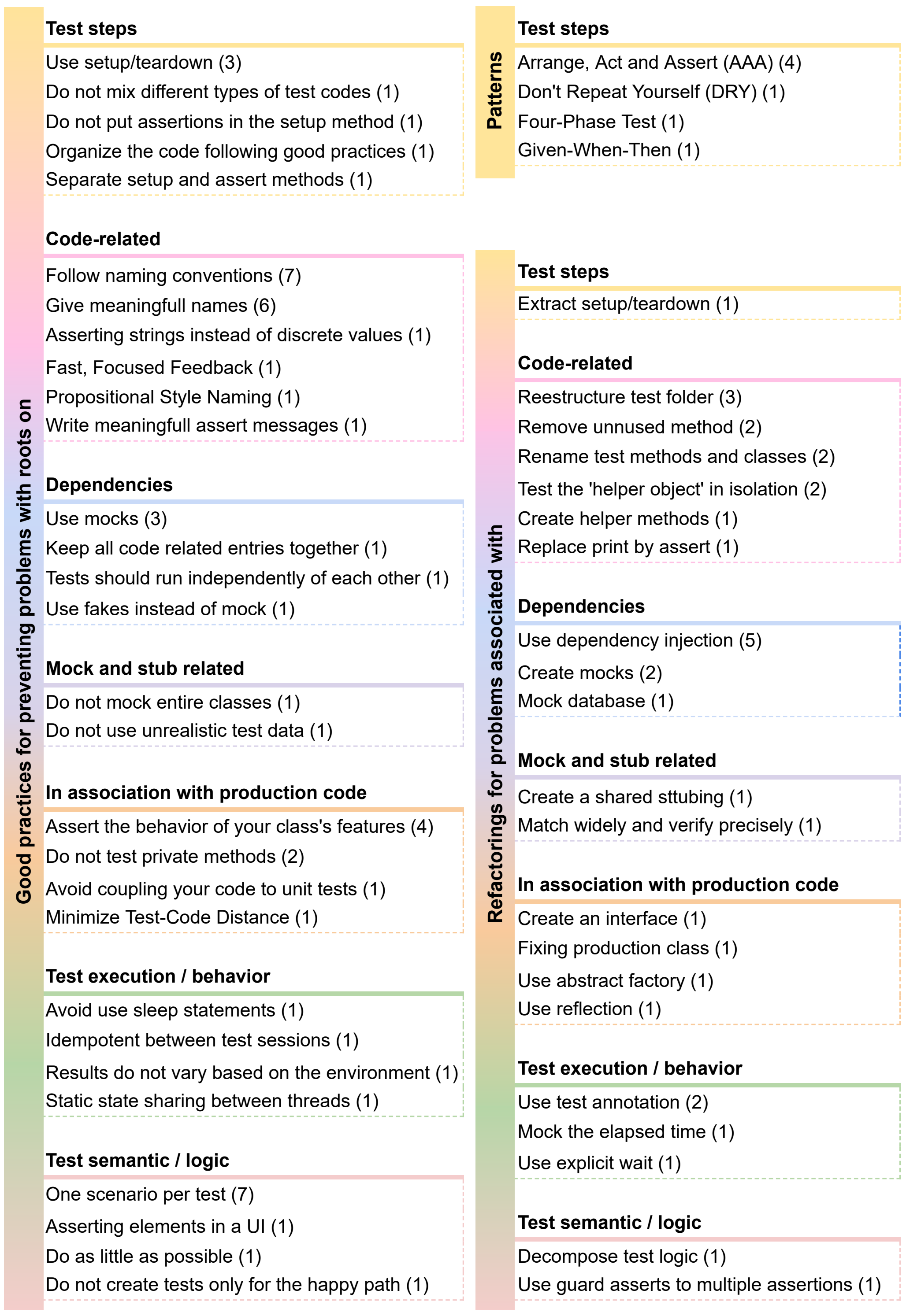
In RQ_3, we investigated the solutions for handling test smells that developers suggest in the Stack Exchange network. We analyzed the actions suggested in the top answers of each discussion, resulting in 33 answers in the *Fix* category, seven answers in the *Capture* category, 70 answers in the *Explain* category, and one answer in the *Ignore* category.

In addition, we extracted the code-based answers, tools, and documents suggested in the answer topics to support developers in fixing test smells. The answers categorized into the *Fix* category suggested 40 test code refactorings for fixing test smells. Less than half of the answers (13; 39.4%) in this category presented code-based refactoring recommendations, i.e., the answers included code samples. In comparison, 32 answers (41.5%) categorized into the *Explain* and *Capture* categories presented 64 patterns for organizing the test code and good practices for preventing test smells. Some answers (10; 12.9%) presented examples of using patterns and good practices. The only answer in the *Ignore* category discussed the trade-offs of refactoring a test method with many assertions to fix a test smell or keeping the assertions as documentation.

Next, we analyzed the solutions proposed in the answers to remove duplicates and group similar answers according to the developers' definitions. Figure 8.8 presents 54 solutions classified into three categories: i) *Good practices for preventing problems with roots on*, ii) *Patterns*, and iii) *Refactorings for problems associated with.* The colors represent the categories of test code problems from Figure 8.7. The patterns, good practices, and test code refactorings aim to solve them.

The *Patterns* category refers to patterns to structure the test code, avoiding issues in the test steps. Most solutions suggested organizing the test methods according to the AAA pattern (BECK, 2003; KHORIKOV, 2020). This pattern organizes the test methods into three steps: (1) setup inputs and targets, (2) act on the target behavior, and (3) assert expected outcomes. It is also called the *Given-When-Then* pattern in the *Behavior Driven Development (BDD)* process (SMART, 2014; KORHONEN, 2020). The *Four-Phase Test* pattern adds one step into that pattern: (4) reset to its pre-setup state (MESZAROS, 2011).

The *Good practices for preventing problems with roots on* category refers to good



Good practices for preventing problems with roots on

Test steps

- Use setup/teardown (3)
- Do not mix different types of test codes (1)
- Do not put assertions in the setup method (1)
- Organize the code following good practices (1)
- Separate setup and assert methods (1)

Code-related

- Follow naming conventions (7)
- Give meaningfull names (6)
- Asserting strings instead of discrete values (1)
- Fast, Focused Feedback (1)
- Propositional Style Naming (1)
- Write meaningfull assert messages (1)

Dependencies

- Use mocks (3)
- Keep all code related entries together (1)
- Tests should run independently of each other (1)
- Use fakes instead of mock (1)

Mock and stub related

- Do not mock entire classes (1)
- Do not use unrealistic test data (1)

In association with production code

- Assert the behavior of your class's features (4)
- Do not test private methods (2)
- Avoid coupling your code to unit tests (1)
- Minimize Test-Code Distance (1)

Test execution / behavior

- Avoid use sleep statements (1)
- Idempotent between test sessions (1)
- Results do not vary based on the environment (1)
- Static state sharing between threads (1)

Test semantic / logic

- One scenario per test (7)
- Asserting elements in a UI (1)
- Do as little as possible (1)
- Do not create tests only for the happy path (1)

Patterns

Test steps

- Arrange, Act and Assert (AAA) (4)
- Don't Repeat Yourself (DRY) (1)
- Four-Phase Test (1)
- Given-When-Then (1)

Refactorings for problems associated with

Test steps

- Extract setup/teardown (1)

Code-related

- Reestructure test folder (3)
- Remove unnused method (2)
- Rename test methods and classes (2)
- Test the 'helper object' in isolation (2)
- Create helper methods (1)
- Replace print by assert (1)

Dependencies

- Use dependency injection (5)
- Create mocks (2)
- Mock database (1)

Mock and stub related

- Create a shared stubing (1)
- Match widely and verify precisely (1)

In association with production code

- Create an interface (1)
- Fixing production class (1)
- Use abstract factory (1)
- Use reflection (1)

Test execution / behavior

- Use test annotation (2)
- Mock the elapsed time (1)
- Use explicit wait (1)

Test semantic / logic

- Decompose test logic (1)
- Use guard asserts to multiple assertions (1)

Figure 8.8: Suggestions of patterns for code organization, good practices, and test code refactorings for preventing and fixing test smells.

practices covering the top test problems presented in Figure 8.7. The *Test Steps* category presents good practices for organizing the test code. The *Code-related* category suggests practices to provide useful information while naming variables and test methods and classes to facilitate the identification of failures caught by the tests. The *Dependencies* category suggests practices for handling dependencies among tests and with external resources. The *Mock and stub related* category brings insights into the correct usage of the mocking frameworks. The *In association with production code* category suggests separating the responsibilities to avoid coupling between test and production codes. The *Test execution/behavior* category aids in developing tests for async tasks of the production code. The *Test semantic/logic* category presents good practices to avoid developing naive test cases and cover as many paths as possible.

Similarly, the *Refactoring for problems associated with* category refers to test code refactorings to solve the seven top problems presented in Figure 8.7. The *Test steps* category presents only one refactoring to extract common arrange code in test methods into setup/teardown methods. The *Code-related* category presents refactorings to deal with code duplication and bad naming. The *Dependencies* category suggests using mocks or dependency injection to handle external dependencies. The *Mock and stub related* category presents refactorings to remove duplication related to mocks and ensure well-designed asserts. The *Association with the production code* category presents refactorings in the production class that can require adaptations in the test code. The *Test execution/behavior* category presents refactorings to handle async tasks by using specific features of the testing or mocking frameworks. Lastly, the *Test semantic/logic* category presents refactorings to decompose the logic in test methods, reducing their complexity.

To guide the adoption of testing patterns, good practices, and test code refactorings, the developers pointed out 32 resources in their answers. Most resources are blogs and books (5; 21.7% each) that present test anti-patterns and refactoring strategies to fix them. In addition, the developers suggested mocking frameworks (7; 30.4%) and pointed out the documentation for constructs of testing and mocking frameworks (4; 17.4%). Some developers also suggested online courses and videos (1; 4.4% each).

Finding 3: Developers more often discuss good practices and test patterns than code-based refactoring recommendations.

8.3.5 Test behavior (RQ4)

In RQ$_4$, we analyzed whether developers are concerned with keeping the test code behavior after performing refactorings to fix test smells. From 101 discussions, only eight (7.9%) addressed test code behavior. In most discussions, the developers are interested in evolving legacy codes, understanding how to perform the refactoring phase of TDD, or improving the test code quality.

After reading Martin Fowler's book (FOWLER, 1999), the developer decided to refactor the test code to organize and remove redundant code. The developer linked refactoring production and test codes, asking how to test the test code refactorings (STACKOVERFLOW, 2009b). As a strategy to keep test code behavior, the answer suggests:

> [...] The trick with complex refactoring of test code is to be able to run the tests against the system under test and get the same results. [...] (STACKOVERFLOW, 2009b)

In another discussion, the developer brings up definitions of refactoring and how to ensure that refactorings do not break the code behavior. Following those definitions, the developer's interest lies in understanding how to refactor smelly test codes and whether creating meta-tests helps keep the test code behavior (STACKOVERFLOW, 2015). The answer suggests:

> When modifying tests, keep the SUT (System Under Test) unchanged. Tests and production code keep each other in check, so varying one while keeping the other locked is safest. [...] (STACKOVERFLOW, 2015)

Finding 4: Discussions on how to keep the test code behavior suggest 1) locking the production code while modifying the tests and 2) checking the results of the tests before and after performing the refactoring. There is no suggestion of tools or strategies to make this process more trustworthy.

8.4 DISCUSSION

This section discusses and interprets the results of each RQ we addressed in this study. Besides, this section points out some existing gaps.

In RQ$_1$, we raised the challenges developers face regarding test code problems and the solutions to cope with them. Developers have to overcome the barrier of (1) convincing management and development teams of the implications of test smells for the test code quality, (2) co-evolving the production and test codes, and (3) keeping themselves up-to-date about the new constructs of programming languages and testing frameworks, as they emerge. Although all the discussions occurred after the first catalogs of test smells (DEURSEN *et al.*, 2001; MESZAROS; SMITH; ANDREA, 2003), we found that developers commonly ask for others' perceptions and practical experience on test code problems and proven solutions. The discussions on the Stack Exchange network highlight a gap between industry and academia, indicating the importance of effectively disclosing literature findings.

In RQ_2, we classified 54 test smells into eight high-level categories of test code problems based on (GAROUSI; KÜÇÜK, 2018). In addition, we publicized the test smell definitions through an online catalog (DATA..., 2022) to help developers understand, prevent, detect, and refactor test smells. Practitioners could use the catalog to educate themselves on test smell concepts. To evolve the catalog, we invited researchers and developers to submit pull requests to update the website with test smells and their definitions.

In RQ_3, we listed the solutions for test code problems and classified them into good practices, test patterns, and test code refactorings. Most solutions explain how to deal with the test code problem but do not demonstrate them. In addition, one good practice or test code refactoring could prevent or refactor more than one test smell. As we could not establish a link between the good practices and test code refactorings to the specific test smells, we linked the good practices and test code refactorings to the top test code problems they aim to solve. Hence, exploring good practices and test code refactorings to fix test smells is essential. Most solutions pointed to blogs, books, documentation, and testing frameworks. Despite the research community's efforts in developing tools for handling test smells and other problems in the test code, the solutions of the Stack Exchange network did not point to any of them. Besides proposing new tools, academic studies should evaluate their usefulness in real-world contexts.

In RQ_4, we observed developers are concerned with keeping the test code behavior after the refactoring. However, no well-defined strategies or tools to aid developers in refactoring test code. It can lead developers to skepticism, missing an opportunity to improve test code quality. Therefore, researchers could focus on providing simple oracle mechanisms for test refactorings (e.g., mutation testing), or automating the catalog of refactorings in a static analyzer.

8.5 THREATS TO VALIDITY

Internal validity. We filtered the discussions related to the test code refactorings and test smells by applying a search string to the tags of each post. As developers can use different tags other than the ones we considered in our search string, we may have missed some discussions in exchange for reducing the number of non-relevant discussions returned with the search string. In addition, we only analyzed the accepted and top-voted responses for each discussion to analyze only the relevant answers that provided a refactoring action or explanation for the problems raised on the Stack Exchange network.

External validity. There are several technology-based question-and-answer websites on the Stack Exchange network. For this study, our scope focused on the top 3 websites (*Stack Overflow*, *Software Engineering*, and *Code Review*), encompassing various programming-related topics. In addition, we applied a search string to the discussions' tags to limit our analysis to test code refactorings performed with Java programming language and the JUnit testing framework.

Conclusion validity. We performed a peer review process to mitigate bias during the selection and classification of the discussions. First, we selected a set of potential discussions, and three researchers classified them independently. We achieved an agreement level of 0.61, following Kappa statistics. Also, the selection and classification processes involved discussions aiming to solve any potential conflicts.

Construct validity. To measure the popularity of a question and an answer, we considered counts of the number of scores on the posts. In addition, we did not consider the period of the questions (e.g., when the developers posted the questions). In our analysis, we did not distinguish between questions based on time, so there are no new questions with low counts of the number of scores and views.

8.6 CONCLUSION

This chapter presented an empirical study we carried out to investigate the discussions about test smells and test code refactoring on the Stack Exchange network. We aimed to leverage knowledge about the corrective actions developers take to deal with them and the main challenges developers face to correct test smells.

To accomplish our goal, we sorted the discussion topics into 30 categories. Each category describes questions developers ask when they face any issue during test code evolution. The yielded results indicate most topics lie in the *Understanding which code structures to test (12)* and *Understanding test smells (12)* categories. We observed that developers are interested in others' perceptions and hands-on experience handling test code issues. In addition, there is an indication that developers often ask whether test smells or anti-patterns are either good or bad testing practices rather than code-based refactoring recommendations.

Chapter 9

TEST SMELL PREVENTION GUIDELINES

The empirical studies carrying out this thesis on the experience of developers (core and peripherals), we could observe that, during the creation of test classes, developers might even be aware of what test smells are, but they still need to learn how to deal with, avoid, or remove these test smells (CAMPOS; ROCHA; MACHADO, 2021). Therefore, as the next step in this thesis, we have designed guidelines to help software developers avoid inserting up to six types of test smells, most recurrent between in previous empirical studies, when creating and maintaining test cases, especially in scenarios involving software evolution.

The guidelines were developed based on the outcomes of the preceding empirical studies we conducted in the chapters 4, 5, 6, 6, 7, and 8. Based on the shared developer knowledge and professional experiences, we have created a set of guidelines to help developers avoid inserting test smells while creating unit test classes. The guidelines include suggestions for avoiding inappropriate practices and identifying test smells to be avoided.

The remainder of this chapter is structured as follows. Section 9.1 introduces the test smell prevention guidelines. Section 9.2 reports on the design of the empirical evaluation. Section 9.3 presents and discusses the key results of the empirical evaluation. In Section 9.4, we discuss the main threats to validity, and in Section 9.5, we conclude this Chapter.

9.1 TEST SMELLS PREVENTION GUIDELINES

9.1.1 Test code

To ensure that the test cases are of the highest quality, it is essential to check whether our test code contains structures that may negatively affect the effectiveness of our tests.

The following structures can be harmful to test code and may be indicators of the presence of test smells.

- A test method that contains multiple assertion statements without an argument as a parameter;
- A test method that contains one or more control statements, i.e., `if`, `switch`, conditional expression, `for`, `foreach`, or `while` statements;
- Method with more than one assertion statement with the same parameters;
- Method contains an instance of a File class without calling the methods `Exists()`, `isFile()` or `noExists()` methods of the object;
- Method that invokes the `Thread.sleep()` method;
- Method does not contain a single assertion statement and `@Test (expected)` annotation parameter.

9.1.2 Test smells prevention

We next provide strategies to cope with six test smell types.

9.1.2.1 Assertion Roulette (AR)

Detection: A test method that contains multiple assertion statements without an argument as a parameter.
Example: Listing 9.1 presents a test method containing *AR* (line 14). The example presents a method of test class `AuthenticationTests` of the `hsweb-framework`.[1].
Preventing AR:

Strategy 1: Splitting into several methods into others to avoid multiple assertions within the same test method;
Strategy 2: Including the explanation parameter;
Strategy 3: Put the setup into a specific method.

[1]Available at https://github.com/hs-web/hsweb-framework project

```
public class AuthenticationTests {

    @Test
    public void testInitUserRoleAndPermission() {
        Authentication authentication = builder.user("{\"id\":\"admin\",\"username\":\"admin\",\"name\":\"
             ↪ Administrator\",\"userType\":\"default\"}")
                .role("[{\"id\":\"admin-role\",\"name\":\"admin\"}]")

                .permission("[{\"id\":\"user-manager\",\"actions\":[\"query\",\"get\",\"update\"]" +
                        ",\"dataAccesses\":[{\"action\":\"query\",\"field\":\"test\",\"fields
                             ↪ \":[\"1\",\"2\",\"3\"],\"scopeType\":\"CUSTOM_SCOPE\",\"type\":\"DENY_FIELDS
                             ↪ \"}]}]")

                .build();

        //test user
        assertEquals(authentication.getUser().getId(), "admin");
        assertEquals(authentication.getUser().getUsername(), "admin");
        assertEquals(authentication.getUser().getName(), "Administrator");
 }
```

Listing 9.1: Code snippet with an AR test smell.

9.1.2.2 Conditional Test Logic (CTL)

Detection: A test method that contains one or more control statements, i.e., if, switch, conditional expression, for, foreach, or while statements.

Example: Listing 9.2 presents a test method containing *CTL* (lines from 8 to 10). The example presents a method of test class `FastBeanCopierTest` from the `hsweb-framework` project.

Preventing CTL:

Strategy: Splitting the method into more methods to reach the conditional structures.

9.1.2.3 Duplicate Assert (DA)

Detection: In the test method, there is more than one assertion statement with the same parameters.

Example: Listing 9.3 presents a test method containing *DA* (lines from 9 to 13). The example presents a method of test class `HashMapTwoFactorTokenManagerTest` from the `hsweb-framework` project.

Preventing DA:

Strategy: Splitting the original method into more test methods for each new value that the variable assumes.

```java
public class FastBeanCopierTest {
    @Test
    public void testProxy() {
        AtomicReference<Object> reference=new AtomicReference<>();

        ProxyTest test = (ProxyTest) Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),
                new Class[]{ProxyTest.class}, (proxy, method, args) -> {
                    if (method.getName().equals("getName")) {
                        return "test";
                    }

                    if (method.getName().equals("setName")) {
                        reference.set(args[0]);
                        return null;
                    }

                    return null;
                });
    }
}
```

Listing 9.2: Code snippet with an CTL test smell.

```java
public class HashMapTwoFactorTokenManagerTest {
    HashMapTwoFactorTokenManager tokenManager = new HashMapTwoFactorTokenManager();

    @Test
    @SneakyThrows
    public void test() {
        TwoFactorToken twoFactorToken = tokenManager.getToken("test", "test");

        Assert.assertTrue(twoFactorToken.expired());
        twoFactorToken.generate(1000L);
        Assert.assertFalse(twoFactorToken.expired());
        Thread.sleep(1100);
        Assert.assertTrue(twoFactorToken.expired());
    }
}
```

Listing 9.3: Code snippet with an DA test smell.

9.1.2.4 Resource Optimism (RO)

Detection: Test method contains an instance of a File class without calling the methods *exists()*, *isFile()* or *noExists()* methods of the object.

Example: Listing 9.4 presents a test method containing *RO* (Lines from 1 to 15). The example presents a method of test class `LocalFileConfigRepositoryTest` from the `apollo` project.

Preventing RO:

```java
@Test
public void testLoadConfigWithLocalFileAndFallbackRepo() throws Exception {
  File file = new File(someBaseDir, assembleLocalCacheFileName());

  String someValue = "someValue";

  Files.write(defaultKey + "=" + someValue, file, Charsets.UTF_8);

  LocalFileConfigRepository localRepo = new LocalFileConfigRepository(someNamespace, upstreamRepo);
  localRepo.setLocalCacheDir(someBaseDir, true);

  Properties properties = localRepo.getConfig();

  assertEquals(defaultValue, properties.getProperty(defaultKey));
}
```

Listing 9.4: Code snippet with an RO test smell.

Strategy 1: Using abstractions for the resource (e.g., mock);
Strategy 2: Creating the resource using the setup method;
Strategy 3: Use JUnit resources to handle temporary files;

9.1.2.5 Sleepy Test (ST)

Detection: A test method that invokes the *Thread.sleep()* method.

Example: Listing 9.5 presents a test method containing *ST* (line 12). The example presents a method of test class `DefaultTimeoutMapTest` from the `pache/camel` project.

```java
1  public class DefaultTimeoutMapTest {
2
3      @Test
4      public void testDefaultTimeoutMapPurge() throws Exception {
5          DefaultTimeoutMap<String, Integer> map = new DefaultTimeoutMap<>(executor, 100);
6          map.start();
7          assertTrue(map.currentTime() > 0);
8          assertEquals(0, map.size());
9          map.put("A", 123, 50);
10         assertEquals(1, map.size());
11
12         Thread.sleep(250);
13         if (map.size() > 0) {
14             LOG.warn("Waiting extra due slow CI box");
15             Thread.sleep(1000);
16         }
17
18         assertEquals(0, map.size());
19
20         map.stop();
21     }
22 }
```

Listing 9.5: Code snippet with an ST test smell.

Preventing ST:

Strategy 1: Use an intelligent waiting library (e.g., Awaitility);
Strategy 2: Make the request asynchronous (e.g., mock);
Strategy 3: Separate it in a method with a test step in a more indicative place;
Strategy 4: Ordering the tests' execution, adding tests containing thread. Sleep command at the end of the test suite.

9.1.2.6 Unknown Test (UT)

Detection: A test method does not contain a single assertion statement and **@Test** (expected) annotation parameter.
Example: Listing 9.6 presents a test method containing *UT* (line from 2 to 15). The example presents a method of test class `RedisUserTokenManagerTest` from the `hsweb-framework` project.

```
1 public class JooqXMLTest extends BaseJooqTest {
2 @Test
3     public void testExecute() {
4         ProducerTemplate producerTemplate = context.createProducerTemplate();
5         producerTemplate.sendBody(context.getEndpoint("direct:execute"), ExchangePattern.InOut, "empty");
6     }
7 }
```

Listing 9.6: Code snippet with an UT test smell.

Preventing UT:

Strategy 1: Including an assertion in the test method;
Strategy 2: Removing this test method depends on its purpose (or lack of purpose).

We made available a supplementary material website hosted on GitHub with all the details about the proposed guidelines and all the material used in the empirical evaluation.[2] In addition, Appendix B includes the forms used in the empirical evaluation. For more tips and examples on preventing test smells during software development, check our website for additional resources (CAMPOS; MARTINS; MACHADO, 2023).

[2]⟨https://github.com/dhennyacampos/testsmellspreventionguidelines⟩

9.2 EMPIRICAL EVALUATION

In this section, we will describe the design of our empirical evaluation, aimed at analyzing the effectiveness of the guidelines.

As illustrated in Figure 9.1, the procedure began with selecting the participants. The selected participants were developers from the industry, and then we classified the developers as core and peripheral. Next, we provided the participants with a detailed session on conducting the evaluation, followed by completing the characterization form. We classified the developers as core and peripheral on the question "Would you say that you are responsible for critical components or modules of the project?". After the classification, the developers were divided into two groups (With and without guidelines). Then, we carried out the actual evaluation tasks and finally asked the developers to fill in a feedback form.

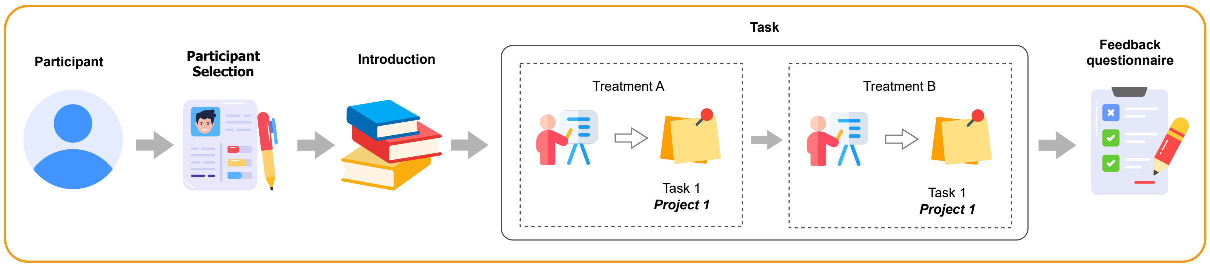


Figure 9.1: Empirical evaluation overview

9.2.1 Research Questions

RQ1: How useful are the proposed guidelines for creating test cases? We employed the Technology Acceptance Model (TAM) (DAVIS, 1989) to address this question. There are other acceptance methods in the literature, but TAM was the most appropriate for our context. TAM is a theory within information systems that seeks to comprehend how developers of new technology accept and use the technology, which in our context consists of a set of guidelines for preventing the insertion of test smells. TAM encompasses three main factors (BABAR; WINKLER; BIFFL, 2007): *Perceived Usefulness (U)*: indicates the degree to which a person believes that using a particular system would improve their work performance; *Perceived Ease-of-Use (E):* indicates the degree to which a person believes that using a particular system would be effortless, and *Self-Predicted Future Use (S):* indicates the degree to which a person believes that they would use a system in the future.

RQ2: What were the developers' main challenges when developing the test cases? In this RQ, we aimed to determine the main challenges the developers came across when creating test cases using test smell prevention guidelines. We used the CALCULATOR project as a key subject in the experiment.

9.2.2 Empirical Evaluation Overview

Target Audience: We selected software developers with experience in Java and the JUnit4 testing framework. We selected 12 people through convenience sampling, i.e., people from our network of contacts who we invited via social networks such as LinkedIn, WhatsApp, or email. Of the 12 developers, 5 had less than one year of experience, another 5 had between one and less than five years of experience, 1 had between five and less than ten years of experience, and 1 had more than ten years of experience with the JUnit framework.

Participants' characterization Form: The participants completed both the consent and characterization forms. The purpose was to examine their background, education level, and expertise in programming and testing.

Participants' feedback Form: The feedback form was structured in two parts. In the first part, the developers filled in an evaluation of their perceptions of the guideline. The questions in this evaluation were based on the TAM. In the second part, the developers shared their perceptions of the main challenges faced during the execution of the experiment, offered suggestions for improvements, and talked about the advantages of using guidelines when creating test cases.

Pilot study: The pilot study was conducted with two participants. Both deal with test case design and are specialists in test smells and test code refactoring. We needed to measure the time they would spend carrying out the experiment activities (see Table 9.1), and analyze the questionnaire to verify that the questions were clear and thorough. The pilot study answers were not considered in the final evaluation.

Initially, we intended to employ two projects (CALCULATOR and E-COMMERCE), one to each group, and we intended to conduct a crossover design study. However, during the pilot study, we realized that the time needed to complete each project's activities exceeded 3 hours. With this in mind, we focused only on the CALCULATOR project designed to validate the guidelines' activities.

Table 9.1: List of tasks the developers have completed

Task	Description
Task #1	Testing the edge values for calculations
	Objective: Create test cases to thoroughly test the divide method of the MathOperations class to verify the edge values for the division operation. Test steps: Considering the class MathOperations, create a test the class named MathOperationsTest for the scenarios: Division of positive integers (dividend >divisor). Division of negative integers (dividend <0, divisor >0). Division by zero (divisor == 0)
Task #2	Reading from an External Database File
	Objective: Create test cases for production classes that involve reading data from external files, such as a database. Test steps: Considering the IOHandler class, please create an IOHandlerTest class to verify the scenarios: Test the IOHandler ability to read data from the external database file Verify that the content read from the file matches the expected format operand operator operand, e.g., 1 + 5
Task #3	Testing parallel calculations
	Objective: Create a test case that verifies the evaluateOperation method of the MathOperations class correctly evaluates mathematical operations in parallel and produces the expected results. Test steps: Considering the class MathOperations, create a test class named MathParallelOperationsTest for the scenario: Create a thread pool with a fixed number of threads. Create a task to execute the evaluateOperation method in parallel with the provided mathematical operation. Verify whether the result obtained from the completed task is correct

Oracle Definition: We developed an *oracle* with the correct task answers. Experts in test smells developed the oracle and have at least three years of experience in this field. As we mentioned above, we selected the `Calculator` project for this experiment. The oracle was used to verify the creation of the test classes and ensure that the six test smells under study *(AR, CTL, DA, RO, ST, UT)* are not inserted. The oracle can be found on our supplementary material website.

Data Collection: To implement the test cases, the participants were free to select the IDE. We only asked the developers not to use ChatGPT or similar tools to create the test cases. After creating the test classes, we checked that the developers had not inserted any test smells during the creation. We used the *JNose Test* tool to detect whether test smells were inserted. After running the tool, the output file with the result shows whether or not the test class contains any of the test smells analyzed. We used the *JNose Test* tool because it detects all test smells and their exact location in the class (line, block, method, or class).

9.2.3 Participants' characterization

In the characterization form, we asked the participants about their background experience. We only selected developers with experience with the JUnit framework to ensure valid results. Of the 12 developers, 41.7% (5) have less than 1 year of experience, 41.7% (5) have between 1 and 5 years of experience, 8.3% (1) have between 5 and 10 years of experience, 8.3% (1) have over years of experience with the JUnit framework. Regarding their engagement in software testing tasks in their daily activities, two developers stated that they perform tests frequently ($\cong$ 75%), five stated that they do so occasionally ($\cong$ 50%), four indicated rarely ($\cong$ 25%). One developer stated that they never create test cases.

We also leveraged information such as the number of commits each participant commonly made (in a given period of time), whether they were responsible for critical modules, and the frequency with which they carried out test cases. We asked the participants to consider the software development projects they are currently involved in. Such data would serve to classify the participants into core or peripheral developers.

After categorizing them into core and peripheral developers, we could associate their awareness of test smells with their experiment grouping. Figure 9.2 shows a big picture of such results. Six out of 12 participants know what test smells are, three categorized as core developers, and three as peripheral developers. Three peripheral developers do not know what test smells are. One core developer and one peripheral developer know what test smells are but do not work with them. One peripheral developer has never heard of test smells.

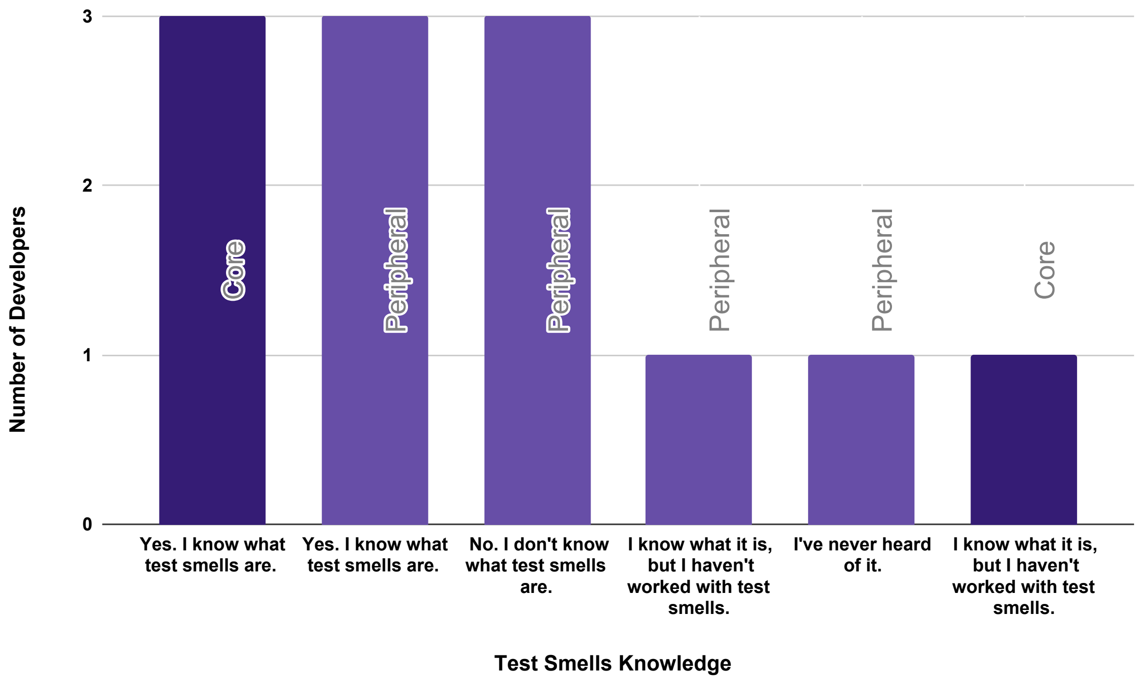


Figure 9.2: Developers' knowledge of test smells

Then, we split the participants into two groups: Group 1 (carrying out the tasks with guideline support) and Group 2 (carrying out the tasks without guideline support). Each group contains 5 participants. Figure 9.3 shows an overview of the groups and the treatments used by each group.

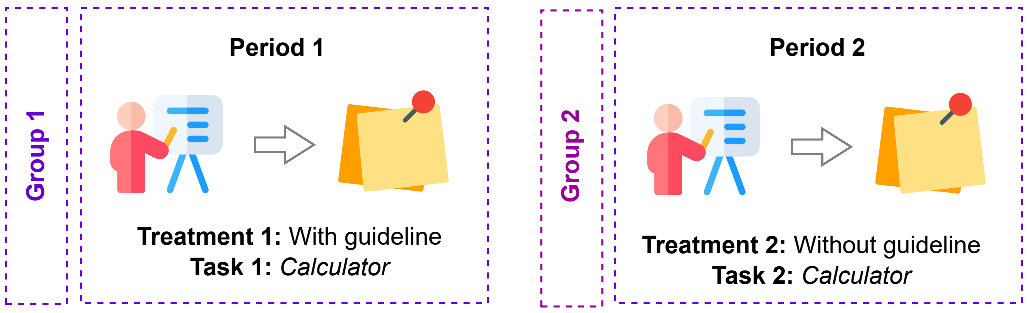


Figure 9.3: Distribution of treatments and tasks

9.3 RESULTS

9.3.1 Usefulness of the proposed guidelines (RQ1)

We employed the TAM method to evaluate how practical and useful the proposed test smell prevention guidelines would be.

The participants filled out a form that explored their perceptions of the guidelines. The questions on the form were designed based on the TAM (Table 9.2), and the answers adopted a six-point Likert scale. This scale was chosen to indicate the likelihood that a statement about the usefulness and ease of use of the test smell prevention guideline is true, ranging from "Extremely unlikely" (1) to "Extremely likely" (6). The six-point scale was chosen to avoid undecided answers.

Table 9.2: Questionnaire used to assess the factors *U, E* and *S*.

Questions about *perceived usefulness (U)* of the guidelines:	
U1.	By using the guidelines in my work, I would be able to create tests faster and with good programming practices.
U2.	Using the guidelines would improve my performance when creating test cases.
U3.	Using the guidelines would increase my effectiveness in creating test cases.
U4.	Using guidelines would make it easier to create test cases.
U5.	I would find the guidelines useful for creating test cases.
U6.	The guidelines used in the experiment are easy to understand and apply.
U7.	Guidelines can help improve the readability and comprehensibility of tests.
Questions about *ease of use (E)* of the guidelines:	
E1.	The guidelines would be easy to use.
E2.	The use of the guidelines is clear and understandable.
E3.	The use of guidelines is applicable to different types of projects and development contexts.
E4.	It's easy to create test cases using the guidelines.
Questions about *self-predicted future use (S)* of the guidelines:	
S1.	Assuming that the guidelines are available in my work, I anticipate using them regularly in the future.
S2.	I would rather use the guidelines than not.

Figure 9.4 shows the results of each question associated with the factors *perceived usefulness (U), ease of use (E),* and *self-predicted future use (S)* of the test smell prevention guidelines. To examine how developers' opinions influence the acceptance of the guidelines, we compared the results related to each criteria group (U, E, and S).

As part of a reliability analysis, we generated Cronbach's alpha coefficients to assess the internal validity and consistency of the items connected with each variable. Cron-

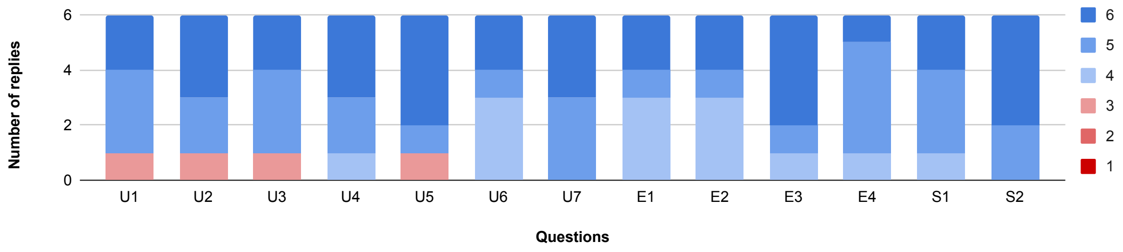


Figure 9.4: Perceived usefulness (U), ease of use (E), and self-predicted future use (S) of the test smell prevention guidelines. The information is showed using a six-point Likert scale, ranging from “Extremely unlikely” (1) to “Extremely likely” (6)

bach’s alpha is a measure of internal reliability that is particularly useful in surveys or questionnaires where multiple questions (items) are designed to evaluate a single construct (factor) (CARMINES; ZELLER, 1979). A Cronbach’s alpha coefficient of 0.61 to 0.70 indicates the lowest limit of acceptability, 0.71 to 0.80 demonstrates item homogeneity, indicating that they measure the same constant, and 0.81 to 1.00 indicates a very reliable measure (CARMINES; ZELLER, 1979; LAITENBERGER; DREYER, 1998).

The results show a Cronbach’s alpha reliability level above 0.92 for the three TAM variables. We obtained a confidence interval 95% as described by Laitenberger and Dreyer (1998).

As Table 9.3 shows, concerning the *U* factor: Variables U1, U2, U3, U4, and U5 all have positive values greater than 0.7, indicating a significant contribution to such a factor. Regarding *E* factor: Variables E1, E2, and E3 all have positive values greater than 0.7, indicating a significant contribution to such a factor as well. U6 shows a strong correlation with the *E* factor, as it has a higher value in this group. The same pattern is observed for the S variables.

9.3.2 Main challenges faced when developing the test cases (RQ2)

This subsection presents and discusses the raised challenges and advantages reported by developers when adopting the guidelines.

Feedback from developers who have used the guidelines.

When developers were questioned about the main challenges they faced when implementing the preventive recommendations in their project, two distinct perspectives emerged. Some participants suggested that more examples be provided to contextualize the standards, stating that more specific examples would be good for teaching. On the other hand, one participant pointed out that the challenge lies in changing behavior when

Table 9.3: Factor Analysis

Variable	Utilization (U)	Use (E)	Use (S)
U1	0,808	0,545	0,220
U2	0,850	0,318	0,364
U3	0,808	0,545	0,220
U4	0,948	-0,137	-
U5	0,968	0,145	0,191
U6	0,238	0,964	0,112
U7	-0,108	-	-0,990
E1	-0,499	0,821	-0,173
E2	0,238	0,964	0,112
E3	0,238	0,964	0,112
E4	0,602	0,565	-0,282
S1	0,673	-	
S2	0,790	0,111	0,154

writing and executing tests, suggesting that the guidelines help improve test code quality but require frequent application to eliminate harmful habits.

The analysis highlights that the most significant challenges are related to changing behavior when writing and executing tests. The guidance helps improve test code quality, but there is a need to apply these guidelines regularly to overcome harmful habits when creating test code.

Another significant observation concerns the need to reduce the guideline size, as it is too long and can be time-consuming to read and understand, particularly for those who are not fluent in English. These analyses emphasize the significance of taking into account the clarity of examples. This information can help improve the guidelines. This suggests reviewing and simplifying the guidelines to make them more accessible to a broader audience.

Advantages of using the test smell prevention guideline.

In response to the question regarding the advantages of following guidelines, we raised interesting points:

- **Test standardization:** The developers mentioned that the standardization of tests is an advantage. This indicates that the guidelines help to establish a standard set of practices, ensuring consistency in the tests performed.

- **Clear and easy-to-remember messages:** The developers claim that the guidelines are simple and share clear, easy-to-remember messages, highlighting the importance of clarity and simplicity in the guidelines and making them easier to apply in practice.

- **Implementing the test steps:** The developers' responses highlight that one advantage is that they show how each test step should be implemented. This suggests that the guidelines offer practical guidance, helping users to understand how to execute tests more efficiently.
- **Anticipating possible flaws in creating test cases:** Using guidelines made it possible to anticipate possible flaws when creating test cases. This is crucial to ensure that tests are robust and reliable from the outset, avoiding future problems.
- **A centralized document of best practices:** The developers report that having a centralized list of good practices to consult is an advantage. This suggests that the guidelines serve as a quick reference for the team, facilitating the adoption of good practices.
- **Avoiding individual code structures in a project:** Developers note that guidelines are a good model for a structured project and avoid individual code structures. This implies that the guidelines promote consistency in coding style throughout the project.

In summary, the perceived advantages of the guidelines include standardization, clarity, practical guidance, exploring good practices in creating test cases, and promotion of consistency in the project. These benefits contribute to developing more efficient, robust, and reliable tests.

Feedback from developers who have NOT used the guidelines.

All the developers said that a set of guidelines for carrying out the tests would be beneficial. This is because, after evaluating the developers' responses, we discovered that some areas demand particular knowledge. For example, we observed a lack of knowledge of the Mockito Library, for which there were difficulties in clearly understanding which test should be written.

The qualitative analysis highlights the importance of comprehensive and clear documentation, providing information on prerequisites, possible modifications, examples of use, and step-by-step instructions. In addition, the precise specification of expected results in test cases, especially in contexts such as thread creation and parallel calculations, is essential to avoid ambiguities and ensure more accessible and more accurate understanding by developers.

When analyzing the answers given by the developers about the main obstacles encountered when developing test cases for the CALCULATOR project, we identified the following key issues and challenges:

- **Lack of knowledge of the Mockito library:** One developer reports a need for knowledge of the Mockito library as an obstacle. This highlights the importance of knowing specific tools to create effective test cases.
- **Need to understand the context and objectives of the project from the source code:** One developer states the need to understand the context and objectives of the project directly from the source code. This indicates the importance of a deep understanding of the code to develop relevant tests.
- **Changes in project structure and lack of examples with guidelines:** According to one developer, there needed to be clearer examples using the guidelines. In addition, uncertainty about how much the project could be changed, such as switching from JUnit 4.12 to 5, was also highlighted as an obstacle.
- **Challenges related to the Java programming language:** The Java language was cited as an obstacle, indicating that some participants may have faced specific challenges related to the programming language itself.
- **Difficulties in specific scenarios and use of Threads:** Some developer responses highlighted specific challenges in scenarios such as thread creation, where the specification of the expected return was not entirely clear. This indicates the complexity of dealing with certain aspects of the code.
- **Complexity in checking calculations in parallel:** One developer mentioned the complexity when trying to check calculations in parallel, highlighting the difficulty in making the check more flexible due to the uncertainty in the order in which the values are executed.
- **Difficulty understanding the test to be written:** One developer mentioned that it was difficult to understand the activities requested, indicating a possible challenge in understanding the requirements or specifying the tests.

- **Lack of experience with JUnit:** One developer pointed out that his biggest obstacle was using JUnit, especially since he had not used it for some time. This highlights the importance of familiarity with testing tools.

In summary, the challenges include the lack of knowledge of specific libraries, lack of proper documentation, changes in project structure, language-specific difficulties, complexity in specific scenarios, and the importance of familiarity with testing tools. This information can guide improvements in documentation, requirements communication, and test case development support.

9.4 THREATS TO VALIDITY

Construct validity. Applying the proposed guideline in other programming languages may require adapting specific implementation details, such as syntax and paradigms. This may involve modifying code structures, libraries, or frameworks used in the guideline.

External Validity. We present the methodology of this study in detail that may allow future replications. We selected software testing practitioners with experience with the JUnit framework. Nevertheless, a more significant number of participants would bring more interesting results. The knowledge, experience, interpretations, and timing of these participants may interfere with the discussions and conclusions of this study. Nevertheless, we believe this study's results could indicate a trend in the area. We intend to investigate this topic further with a more considerable number of test smells and projects.

9.5 CONCLUSION

This chapter presented the proposed guideline support together with its empirical evaluation. The results pointed to a positive relationship with the guidelines, with developers recognizing their potential to improve the creation and effectiveness of test cases. The high Cronbach's alpha coefficients demonstrated the internal reliability and consistency of the survey items, reinforcing the validity of the findings.

In addition, feedback from developers who used the guidelines highlighted both challenges and advantages. Challenges included the length of the guidelines and difficulties related to language barriers. On the positive side, developers recognized the benefits of standardized testing, clear messaging, practical guidance, anticipating failures, centralized best practices, and promoting project consistency.

In summary, the findings of this study contribute valuable information about how useful the proposed guidelines could be in practice. The research highlights the need to take into account a variety of developer experiences and perspectives during the process of developing and implementing test guidelines. Even those with experience in JUnit faced challenges when creating test cases. As a prospect for future research, we aim to broaden the range of test smells covered in our guidelines.

Chapter 10

CONCLUSIONS

This thesis has gathered the results of a series of empirical studies, whose main goal was to analyze human factors' impact (mainly in terms of the developers' experience and contribution to software projects) on test code quality from the perspective of test smells.

We raised questions about the current awareness of the test smell concepts. We made the developers think of how the occurrence of such design issues in their test code could impact the overall quality of the product they are developing. At least those software practitioners involved in the empirical studies could then understand how the occurrence of test smells harms software quality, causing poor comprehensibility of test code, poorly written code, difficulties in maintaining and evolving the software, and can lead to possible technical debt.

The combined results of the empirical studies carried out in this thesis could raise a set of practices that software testing practitioners should consider when designing and implementing their test cases. This is the last result of this thesis, proposing and evaluating guideline support to aid the design of test cases with a reduced likelihood of inserting test smells. The qualitative empirical studies aimed to leverage the challenges the developers commonly face in their daily practices and analyze whether guidelines support could yield benefits to the quality of the implemented test cases.

We understand all these empirical studies have limitations, and as such, we would like to pave the way for future work by describing some opportunities for further research in this field. Section 10.1 addresses some of such possibilities. Finally, Section 10.4 highlights the main publications of this thesis.

10.1 OPEN ISSUES

However, other dimensions of knowledge have not been explicitly addressed in this thesis. Studying the effects of motivation and satisfaction could be another topic of interest and worth investigating. Other non-technical factors, such as team dynamics, project management practices and communication styles, cultural background, educational levels, and gender, could also be explored to understand better how different forms of knowledge influence test code quality.

Back to the technical perspective, we listed some key directions that emerged from the study we just carried out:

Training. Developing and evaluating training strategies aimed at developers in the context of designing and maintaining high-quality tests. These strategies can include creating educational materials, practical workshops, and implementing mentoring programs.

Evaluating automated tools. The analysis of the survivability of test smell in open-source projects indicates possible gaps in current refactoring practices, pointing to the need for more proactive strategies. This future work could involve developing or improving existing tools to help developers apply the proposed guidelines. The aim is to reduce the occurrence of specific test smells that reoccur during the refactoring process. This can help identify patterns of continuous improvement or persistent challenges in software development projects.

Frameworks. Expand the scope of the analysis beyond the *JUnit* framework, exploring other testing frameworks widely adopted in various development communities. This expansion will make it possible to obtain more comprehensive and specific insights into testing practices in different technologies and development contexts.

10.2 IMPACTS OF THIS THESIS

Social: It promotes the appreciation of the knowledge and skills in software development teams, encouraging more inclusive collaboration and better communication between professionals with different backgrounds and experiences.

Technological: It creates practical guidelines for preventing “test smells´´ and improving test code quality, which contributes to the development of more robust and sustainable software, improving software maintenance and evolution practices.

Scientific: It fills a gap in the literature on the relationship between developer knowledge and test code quality. offering empirical evidence that can guide future research into human factors in software engineering.

Environmental: Indirectly, by improving the efficiency and quality of software projects, this work can reduce the waste of computing resources and energy used in maintenance processes and rework faulty software.

10.3 THREATS TO VALIDITY OF THE THESIS

Although the studies contribute valuable information about the relationship between developers' knowledge and test code quality, it is essential to recognize potential threats to the validity of the findings. The following threats must be considered:

External Validity. The studies rely on data from open-source projects and practitioners within specific communities. Participants may have different motivation levels or expertise, potentially skewing the understanding of practitioners' awareness and strategies regarding test smells. External factors, such as organizational policies or project-specific constraints, could influence the observed practices and outcomes. To mitigate these two threats, we only use the knowledge of developers on projects and JUnit framework. Findings may not be generalizable to a more expansive population of developers with different backgrounds, experiences, or tool preferences, that is, because Software development practices can differ significantly across cultures and organizations, and the conclusions may not be applicable in settings with distinct cultures.

Internal validity. The studies primarily focus on the *JUnit* testing framework. The use of the *JUnit* testing framework in the studies may mitigate the generalizability of the conclusions to projects that use other testing frameworks. The results may not fully capture the knowledge of testing practices across different frameworks. The studies rely on developers' interpretations, and subjective differences in assessing the severity of test smells could introduce ambiguity in the results. While the studies consider the roles of core and peripheral developers, other aspects of knowledge, such as cultural backgrounds, educational levels, or industry experience, are not explicitly explored. These factors could also play a role in influencing test code quality. Mitigating these internal threats involves careful study design, rigorous data collection methods, and transparency in reporting. Researchers acknowledge potential biases and limitations and take steps to minimize their impact on the internal validity of the study.

Construct validity. The effectiveness of refactoring practices is measured based on removing test smells. While this provides insights into the immediate impact, it may only partially capture the long-term consequences or potential introduction of new issues during refactoring. To mitigate this threat, we selected test cases only with the *Junit* Framework.

Conclusion validity. Conclusions about the influence of developer knowledge on test code quality may be more balanced with the interaction of human aspects. Mitigating these threats to the conclusion involves careful consideration of the scope and limitations of the research and providing differentiated interpretations of the dynamics of software development.

10.4 SCIENTIFIC CONTRIBUTIONS

We next enlist the publications associated with this thesis:

1. **CAMPOS, DENIVAN**; MARTINS, L.; BEZERRA, C.; MACHADO, I. Investigating Developers' Contributions to Test Smell Survivability: A Study of Open-Source Projects. In: Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2023, Campo Grande - MS. VIII Simpósio Brasileiro de Teste de Software Sistemático e Automatizado, 2023.

2. **CAMPOS, DENIVAN**; MARTINS, LUANA ; MACHADO, IVAN . An empirical study on the influence of developers? experience on software test code quality. In: SBQS '22: XXI Brazilian Symposium on Software Quality, 2022, Curitiba Brazil. Proceedings of the XXI Brazilian Symposium on Software Quality. p. 1.

3. **CAMPOS, DENIVAN**; ROCHA, L.; MACHADO, I. Developers' perception on the severity of test smells: an empirical study. In: XXIV Ibero-American Conference on Software Engineering (CIbSE2021), 2021, San Jose - Costa Rica. XXIV Ibero-American Conference on Software Engineering (CIbSE), 2021.

Then, it is worth mentioning some other studies we carried out that contributed to this thesis to a certain extent:

1. MARTINS, LUANA ; **CAMPOS, DENIVAN**; SANTANA, R. ; MOTA JUNIOR, J. ; COSTA, H. ; Ivan Machado . Hearing the voice of experts: Unveiling Stack Exchange communities' knowledge of test smells. In: 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE 2023), 2023, Melbourne, Austrália. 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE 2023), 2023.

2. DAMASCENO, H.; BEZERRA, C.; **CAMPOS, DENIVAN**; 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, v. 11, p. 15, 2023.

3. SANTANA, R.; FERNANDES, D. D.; **CAMPOS, DENIVAN**; ROCHA, L.; MACIEL, R.MACHADO, I. . Understanding practitioners' strategies to handle test smells: a multi-method study. In: Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2021, Joinville - SC. Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2021.

REFERENCES

ALIZADEH, V.; KESSENTINI, M.; MKAOUER, M. W.; CINNEIDE, M. O.; OUNI, A.; CAI, Y. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, v. 46, n. 9, p. 932–961, 2020.

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 (EASE)*. [S.l.]: ACM, 2021. p. 170–180.

ALOMAR, E. A.; PERUMA, A.; MKAOUER, M. W.; NEWMAN, C. D.; OUNI, A. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2395, 2021.

ALOMAR, E. A.; PERUMA, A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. New York, NY, USA: Association for Computing Machinery, 2020. (ICSEW'20), p. 342–349. Available at: ⟨https://doi.org/10.1145/3387940.3392193⟩.

AMMANN, P.; OFFUTT, J. *Introduction to software testing*. [S.l.]: Cambridge University Press, 2016.

AMPATZOGLOU, A.; TSINTZIRA, A.-A.; ARVANITOU, E.-M.; CHATZIGEORGIOU, A.; STAMELOS, I.; MOGA, A.; HEB, R.; MATEI, O.; TSIRIDIS, N.; KEHAGIAS, D. Applying the single responsibility principle in industry: Modularity benefits and trade-offs. In: *Proceedings of the Evaluation and Assessment on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (EASE '19), p. 347–352.

ATHANASIOU, D.; NUGROHO, A.; VISSER, J.; ZAIDMAN, A. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, v. 40, n. 11, p. 1100–1125, 2014.

BABAR, M. A.; WINKLER, D.; BIFFL, S. Evaluating the usefulness and ease of use of a groupware tool for the software architecture evaluation process. In: IEEE. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. [S.l.], 2007. p. 430–439.

BAI, G. R.; PRESLER-MARSHALL, K.; FISK, S. R.; STOLEE, K. T. Is assertion roulette still a test smell? an experiment from the perspective of testing education. In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. [S.l.: s.n.], 2022. p. 1–7.

BANKER, R. D.; DATAR, S. M.; KEMERER, C. F. A model to evaluate variables impacting the productivity of software maintenance projects. *Management Science*, INFORMS, v. 37, n. 1, p. 1–18, 1991.

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: *28th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2012.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, v. 20, n. 4, p. 1052–1094, 2015.

BEAVER, J. M.; SCHIAVONE, G. A. The effects of development team skill on software product quality. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 31, n. 3, p. 1–5, 2006.

BECK, K. *Test-driven development: by example*. [S.l.]: Addison-Wesley Professional, 2003.

BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*. [S.l.: s.n.], 2000. p. 73–87.

BHASIN, T.; MURRAY, A.; STOREY, M.-A. Student experiences with github and stack overflow: An exploratory study. In: IEEE. *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. [S.l.], 2021. p. 81–90.

BIRD, C.; NAGAPPAN, N.; MURPHY, B.; GALL, H.; DEVANBU, P. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox. *Technical report, University of California*, 2010.

BOH, W. F.; SLAUGHTER, S. A.; ESPINOSA, J. A. Learning from experience in software development: A multilevel analysis. *Management science*, Informs, v. 53, n. 8, p. 1315–1331, 2007.

BORGES, H.; VALENTE, M. T. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, Elsevier, v. 146, p. 112–129, 2018.

BOSU, A.; CARVER, J. C. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: Association for Computing Machinery, 2014. (ESEM '14). Available at: ⟨https://doi.org/10.1145/2652524.2652544⟩.

BRAUN, V.; CLARKE, V. Using thematic analysis in psychology. *Qualitative research in psychology*, Taylor & Francis, v. 3, n. 2, p. 77–101, 2006.

CALEFATO, F.; GEROSA, M. A.; IAFFALDANO, G.; LANUBILE, F.; STEINMACHER, I. Will you come back to contribute? investigating the inactivity of oss core developers in github. *Empirical Software Engineering*, Springer, v. 27, n. 3, p. 1–41, 2022.

CAMPOS, D.; MARTINS, L.; BEZERRA, C.; MACHADO, I. Dataset. 8 2023. Available at: ⟨https://figshare.com/articles/dataset/Dataset/23614899⟩.

CAMPOS, D.; MARTINS, L.; BEZERRA, C.; MACHADO, I. Investigating developer contributions to test smell survivability: A study of open-source projects. In: *Proceedings of the VIII Brazilian Symposium on Systematic and Automated Software Testing*. [S.l.: s.n.], 2023.

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*. [S.l.: s.n.], 2022. p. 1–10.

CAMPOS, D.; MARTINS, L.; MACHADO, I. *Test smells prevention guidelines*. 2023. Accessed on 07.14.2022. Available at: ⟨https://github.com/dhennyacampos/testsmellspreventionguidelines⟩.

CAMPOS, D.; ROCHA, L.; MACHADO, I. Developers perception on the severity of test smells: an empirical study. *XXIV Ibero-American Conference on Software Engineering*, 2021.

CANEDO, E. D.; BONIFÁCIO, R.; OKIMOTO, M. V.; SEREBRENIK, A.; PINTO, G.; MONTEIRO, E. Work practices and perceptions from women core developers in oss communities. In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Bari, Italy: Association for Computing Machinery, 2020. p. 1–11.

CANNING, R. G. That maintenance 'iceberg'. *EDP Analyzer*, v. 10, n. 10, p. 1–14, 1972.

CARMINES, E. G.; ZELLER, R. A. *Reliability and validity assessment*. [S.l.]: Sage publications, 1979.

CHEN, Z.; EMBURY, S.; VIGO, M. Evaluating test smells in open-source projects. In: *IFIP-ICTSS 35th International Conference on Testing Software and Systems*. [S.l.: s.n.], 2023.

CHOI, E.; YOSHIDA, N.; KULA, R. G.; INOUE, K. What do practitioners ask about code clone? a preliminary investigation of stack overflow. In: *2015 IEEE 9th International Workshop on Software Clones (IWSC)*. [S.l.: s.n.], 2015. p. 49–50.

CHRISTOPOULOU, A.; GIAKOUMAKIS, E. A.; ZAFEIRIS, V. E.; SOUKARA, V. Automated refactoring to the strategy design pattern. *Information and Software Technology*, Elsevier, v. 54, n. 11, p. 1202–1214, 2012.

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.

CROWSTON, K.; HOWISON, J. The social structure of free and open source software development. *First Monday*, 2005.

CROWSTON, K.; WEI, K.; LI, Q.; HOWISON, J. Core and periphery in free/libre and open source software team communications. In: IEEE. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*. [S.l.], 2006. v. 6, p. 118a–118a.

DABIC, O.; AGHAJANI, E.; BAVOTA, G. Sampling projects in github for MSR studies. In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. [S.l.]: IEEE, 2021. p. 560–564.

DARCY, D. P.; KEMERER, C. F.; SLAUGHTER, S. A.; TOMAYKO, J. E. The structural complexity of software an experimental test. *IEEE Transactions on software engineering*, IEEE, v. 31, n. 11, p. 982–995, 2005.

DATA Collection and analysis. 2022. Accessed on 12.23.2022. Available at: ⟨https://github.com/arieslab/testsmells⟩.

DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, JSTOR, p. 319–340, 1989.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013.

DEMARCO, T.; LISTER, T. *Peopleware: productive projects and teams*. [S.l.]: Addison-Wesley, 2013.

DEURSEN, A.; MOONEN, L. M.; BERGH, A.; KOK, G. *Refactoring Test Code*. NLD, 2001. 92–95 p.

DIBBLE, C.; GESTWICKI, P. Refactoring code to increase readability and maintainability: A case study. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, Evansville, IN, USA, v. 30, n. 1, p. 41–51, oct 2014. ISSN 1937-4771.

EWASCHUK, R.; BEYER, B. Monitoring distributed systems. In: ______. *Site Reliability Engineering: How Google Runs Production Systems*. [s.n.], 2016. Available at: ⟨https://landing.google.com/sre/book.html⟩.

FAGERHOLM, F.; MÜNCH, J. Developer experience: Concept and definition. In: IEEE. *2012 international conference on software and system process (ICSSP)*. [S.l.], 2012. p. 73–77.

FERREIRA, M.; VALENTE, M. T.; FERREIRA, K. A comparison of three algorithms for computing truck factors. In: *Proceedings of the 25th International Conference on Program Comprehension*. Buenos Aires, Argentina: IEEE Press, 2017. (ICPC '17), p. 207–217. Available at: ⟨https://doi.org/10.1109/ICPC.2017.35⟩.

FOWLER, M. *Refactoring: Improving the Design of Existing Code (Object Technology Series)*. illustrated edition. Amsterdam: Addison-Wesley Longman, 1999.

FREIRE, S.; RIOS, N.; PÉREZ, B.; CASTELLANOS, C.; CORREAL, D.; RAMAČ, R.; MANDIĆ, V.; TAUŠAN, N.; LÓPEZ, G.; PACHECO, A. *et al.* How experience impacts practitioners' perception of causes and effects of technical debt. In: IEEE. *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. [S.l.], 2021. p. 21–30.

GAROUSI, V.; KÜÇÜK, B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, Elsevier, v. 138, p. 52–81, 2018.

GAROUSI, V.; PETERSEN, K.; OZKAN, B. Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology*, Elsevier, v. 79, p. 106–127, 2016.

GOMES, F.; SANTOS, E. P. d.; FREIRE, S.; MENDONçA, M.; MENDES, T. S.; SPINOLA, R. Investigating the point of view of project management practitioners on technical debt: A preliminary study on stack exchange. In: *Proceedings of the International Conference on Technical Debt*. New York, NY, USA: Association for Computing Machinery, 2022. (TechDebt '22), p. 31–40.

GOUSIOS, G.; ZAIDMAN, A. A dataset for pull-based development research. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2014. (MSR 2014), p. 368–371. Available at: ⟨https://doi.org/10.1145/2597073.2597122⟩.

GREILER, M.; DEURSEN, A. V.; STOREY, M.-A. Automated detection of test fixture strategies and smells. In: IEEE. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. [S.l.], 2013. p. 322–331.

GUERRA, A.; COLOMBO, R. *Tecnologia da informação: qualidade de produto de software*. Ministério da Ciência e Tecnologia, 2009. Available at: ⟨https://books.google.com.br/books?id=5j1sQwAACAAJ⟩.

HERBSLEB, J. D.; GRINTER, R. E. Architectures, coordination, and distance: Conway's law and beyond. *IEEE software*, IEEE, v. 16, n. 5, p. 63–70, 1999.

HESSE-BIBER, S. N. *Mixed methods research: Merging theory with practice.* [S.l.]: Guilford Press, 2010.

HOZANO, M.; GARCIA, A.; FONSECA, B.; COSTA, E. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology*, Elsevier, v. 93, p. 130–146, 2018.

IEEE. *ANSI/IEEE Std 729-1983 Glossary of Software Engineering Terminology.* New York, NY, USA: [s.n.], 1983.

JACOBSON, I.; LAWSON, H. B.; NG, P.-W.; MCMAHON, P. E.; GOEDICKE, M. *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!* [S.l.]: Association for Computing Machinery and Morgan &amp; Claypool, 2019. ISBN 9781947487277.

JALOTE, P. *An integrated approach to software engineering.* [S.l.]: Springer Science & Business Media, 2012.

JOBLIN, M.; APEL, S.; HUNSEN, C.; MAUERER, W. Classifying developers into core and peripheral: An empirical study on count and network metrics. In: *Proceedings of the 39th International Conference on Software Engineering.* Buenos Aires, Argentina: IEEE Press, 2017. (ICSE '17), p. 164–174. Available at: ⟨https://doi.org/10.1109/ICSE.2017.23⟩.

JUNG, R.; MÄRTIN, L.; JOHANSSEN, J. O.; PAECH, B.; LOCHAU, M.; THÜM, T.; SCHNEIDER, K.; TICHY, M.; ULBRICH, M. Addressed challenges. In: _____. *Managed Software Evolution.* Cham: Springer International Publishing, 2019. p. 21–36.

JUNIOR, N. S.; 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.

KASUNIC, M. *Designing an Effective Survey. Carnegie Mellon University, Software Engineering Institute.* Pittsburgh, PA, 2005. CMU/SEI-2005-HB-004.

KATAOKA, Y.; IMAI, T.; ANDOU, H.; FUKAYA, T. A quantitative evaluation of maintainability enhancement by refactoring. In: *International Conference on Software Maintenance, 2002. Proceedings.* [S.l.: s.n.], 2002. p. 576–585.

KATIĆ, M.; FERTALJ, K. Towards an appropriate software refactoring tool support. In: *WSEAS international conference on applied computer science.* [S.l.: s.n.], 2009. p. 140–145.

KHLEEL, N. A. A.; NEHÉZ, K. Mining software repository: an overview. *Doktoranduszok Fóruma*, p. 108, 2020.

KHORIKOV, V. *Unit Testing Principles, Practices, and Patterns.* [S.l.]: Simon and Schuster, 2020.

KIM, D. J. An empirical study on the evolution of test smell. In: IEEE. *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* [S.l.], 2020. p. 149–151.

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. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, IEEE, v. 40, n. 7, p. 633–649, 2014.

KORHONEN, J. *Automated Model Generation Using GraphWalker Based on Given-When-Then Specifications.* Bachelor's Thesis — Malardalen University, Vasteras, Sweden, July 2 2020. Available at: ⟨http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-49294⟩.

LAITENBERGER, O.; DREYER, H. M. Evaluating the usefulness and the ease of use of a web-based inspection data collection tool. In: IEEE. *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262).* [S.l.], 1998. p. 122–132.

LEE, A.; CARVER, J. C. Are one-time contributors different? a comparison to core and periphery developers in floss repositories. In: IEEE. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* [S.l.], 2017. p. 1–10.

LEHMAN, M.; RAMIL, J. F. Software evolution in the age of component-based software engineering. *IEE Proceedings-Software*, IET, v. 147, n. 6, p. 249–255, 2000.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, IEEE, v. 68, n. 9, p. 1060–1076, 1980.

LEHMAN, M. M.; RAMIL, J. F. Software evolution—background, theory, practice. *Information Processing Letters*, Elsevier, v. 88, n. 1-2, p. 33–44, 2003.

LUO, L. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 2001.

LYU, M. R. (Ed.). *Handbook of Software Reliability Engineering.* USA: McGraw-Hill, Inc., 1996. ISBN 0070394008.

MARSH, R.; BELGUITH, S.; DARGAHI, T. Iot database forensics: an investigation on harperdb security. In: *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems.* [S.l.: s.n.], 2019. p. 1–7.

MARTINS, L.; PONTILLO, V.; COSTA, H.; FERRUCCI, F.; PALOMBA, F.; MACHADO, I. Test code refactoring unveiled: Where and how does it affect test code quality and effectiveness? *arXiv preprint arXiv:2308.09547*, 2023.

MARTINS, L. A.; CAMPOS, D.; SANTANA, R.; JúNIOR, J. M.; COSTA, H. A. X.; MACHADO, I. Hearing the voice of experts: Unveiling stack exchange communities' knowledge of test smells. In: *16th IEEE/ACM International Conference on Cooperative and Human Aspects of Software Engineering (CHASE@ICSE)*. [S.l.]: IEEE, 2023. p. 80–91.

MATSUMOTO, S.; KAMEI, Y.; MONDEN, A.; MATSUMOTO, K.-i.; NAKAMURA, M. An analysis of developer metrics for fault prediction. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. [S.l.: s.n.], 2010. p. 1–9.

MEHTA, M. Date functions. In: *Microsoft Excel Functions Quick Reference*. [S.l.]: Springer, 2021. p. 31–51.

MENS, T.; GUEHéNéUC, Y.-G.; FERNáNDEZ-RAMIL, J.; D'HONDT, M. Guest editors' introduction: Software evolution. *IEEE Software*, v. 27, n. 4, p. 22–25, 2010.

MENS, T.; SEREBRENIK, A.; CLEVE, A. *Evolving Software Systems*. [S.l.]: Springer, 2014.

MESZAROS, G. *xUnit test patterns: Refactoring test code*. [S.l.]: Addison-Wesley Professional, 2007.

MESZAROS, G. *xUnit Patterns - Four-Phase Test*. 2011. Accessed on 12.26.2022. Available at: ⟨http://xunitpatterns.com/Four\\%20Phase\\%20Test.html⟩.

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.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 11, n. 3, p. 309–346, jul 2002. ISSN 1049-331X.

NAGAPPAN, N.; MURPHY, B.; BASILI, V. The influence of organizational structure on software quality. In: IEEE. *2008 ACM/IEEE 30th International Conference on Software Engineering*. [S.l.], 2008. p. 521–530.

OLIVEIRA, R.; MELLO, R. de; FERNANDES, E.; GARCIA, A.; LUCENA, C. Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers. *Information and Software Technology*, Elsevier, v. 120, p. 106242, 2020.

OPENJA, M.; ADAMS, B.; KHOMH, F. Analysis of modern release engineering topics : – a large-scale study using stackoverflow –. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2020. p. 104–114.

PAIVA, T.; DAMASCENO, A.; FIGUEIREDO, E.; SANT'ANNA, C. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, SpringerOpen, v. 5, n. 1, p. 1–28, 2017.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D. Do they really smell bad? a study on developers' perception of bad code smells. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2014. p. 101–110.

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: IEEE. *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. [S.l.], 2016. p. 5–14.

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. Available at: ⟨https://doi.org/10.1007/s10664-022-10207-5⟩.

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: ACM, 2020. (ESEC/FSE 2020), p. 1650–1654.

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.

PERUMA, A. S. A. *What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications*. Thesis — Rochester Institute of Technology, 2018. Available at: ⟨https://repository.rit.edu/cgi/viewcontent.cgi?article=10936&context=theses⟩.

PINTO, G. H.; KAMEI, F. What programmers say about refactoring tools? an empirical investigation of stack overflow. In: *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*. New York, NY, USA: Association for Computing Machinery, 2013. (WRT '13), p. 33–36.

PINZGER, M.; NAGAPPAN, N.; MURPHY, B. Can developer-module networks predict failures? In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. [S.l.: s.n.], 2008. p. 2–12.

POHL, K.; BOCKLE, G.; LINDEN, F. J. v. *Software Product Line Engineering: Foundations, Principles and Techniques*. [S.l.]: Springer-Verlag New York, Inc., 2005.

POSNETT, D.; WARBURG, E.; DEVANBU, P.; FILKOV, V. Mining stack exchange: Expertise is evident from initial contributions. In: *2012 International Conference on Social Informatics*. [S.l.: s.n.], 2012. p. 199–204.

PRESSMAN, R. S.; MAXIM, B. R. *Software engineering: a practitioner's approach*. 9. ed. [S.l.]: McGraw Hill, 2020.

RAHMAN, F.; DEVANBU, P. Ownership, experience and defects: a fine-grained study of authorship. In: *Proceedings of the 33rd International Conference on Software Engineering*. [S.l.: s.n.], 2011. p. 491–500.

RAJLICH, V. Software evolution and maintenance. In: *Future of Software Engineering Proceedings*. [S.l.: s.n.], 2014. p. 133–144.

SALGADO, S. A. D. *Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics*. Master's Thesis (Mestrado Integrado em Engenharia Informática e Computação) — Faculdade de Engenharia, Universidade do Porto, July 22 2020. Available at: ⟨https://repositorio-aberto.up.pt/bitstream/10216/128585/2/412488.pdf⟩.

SANTANA, R.; FERNANDES, D.; CAMPOS, D.; SOARES, L.; MACIEL, R.; MACHADO, I. Understanding practitioners' strategies to handle test smells: a multi-method study. In: *Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2021. p. 49–53.

SANTANA, R.; MARTINS, L.; ROCHA, L.; VIRGÍNIO, T.; CRUZ, A.; COSTA, H.; MACHADO, I. RAIDE: a tool for assertion roulette and duplicate assert identification and refactoring. In: *34th Brazilian Symposium on Software Engineering (SBES)*. [S.l.]: ACM, 2020.

SANTOS, E. P.; GOMES, F.; FREIRE, S.; MENDONçA, M.; MENDES, T. S.; SPINOLA, R. Technical debt on agile projects: Managers' point of view at stack exchange. In: *Proceedings of the XXI Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '22).

SETIA, P.; RAJAGOPALAN, B.; SAMBAMURTHY, V.; CALANTONE, R. How peripheral developers contribute to open-source software development. *Information Systems Research*, INFORMS, v. 23, n. 1, p. 144–163, 2012.

SILVA-JUNIOR, N.; ROCHA, L.; MARTINS, L. A.; MACHADO, I. A survey on test practitioners' awareness of test smells. In: *Proceedings of the XXIII Iberoamerican Conference on Software Engineering (CIbSE)*. [S.l.]: Curran Associates, 2020. p. 462–475.

SMART, J. *BDD in Action: Behavior-driven development for the whole software lifecycle*. [S.l.]: Simon and Schuster, 2014.

SPADINI, D.; PALOMBA, F.; ZAIDMAN, A.; BRUNTINK, M.; BACCHELLI, A. On the relation of test smells to software code quality. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME*. [S.l.]: IEEE, 2018.

SPADINI, D.; SCHVARCBACHER, M.; OPRESCU, A.-M.; BRUNTINK, M.; BACCHELLI, A. Investigating severity thresholds for test smells. In: *17th International Conference on Mining Software Repositories*. [S.l.]: ACM, 2020.

SPÍNOLA, R. O.; ZAZWORKA, N.; VETRO, A.; SHULL, F.; SEAMAN, C. Understanding automated and human-based technical debt identification approaches-a two-phase study. *Journal of the Brazilian Computer Society*, SpringerOpen, v. 25, n. 1, p. 1–21, 2019.

STACKOVERFLOW. *Unit testing Anti-patterns catalogue*. 2008. Accessed on 12.23.2022. Available at: ⟨https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue⟩.

STACKOVERFLOW. *Do I need to test helper/setup methods?* 2009. Accessed on 12.23.2022. Available at: ⟨https://stackoverflow.com/questions/1004866/do-i-need-to-test-helper-setup-methods⟩.

STACKOVERFLOW. *Refactoring and Test Driven Development*. 2009. Accessed on 12.26.2022. Available at: ⟨https://stackoverflow.com/questions/657645/⟩.

STACKOVERFLOW. *Unit test coding standards*. 2010. Accessed on 12.23.2022. Available at: ⟨https://softwareengineering.stackexchange.com/questions/15925/⟩.

STACKOVERFLOW. *If your unit test code "smells" does it really matter?* 2011. Accessed on 12.23.2022. Available at: ⟨https://softwareengineering.stackexchange.com/questions/77313/⟩.

STACKOVERFLOW. *Java: code duplication in classes and their Junit test cases*. 2012. Accessed on 12.29.2022. Available at: ⟨https://stackoverflow.com/questions/10781050/java-code-duplication-in-classes-and-their-junit-test-cases⟩.

STACKOVERFLOW. *How do I refactor unit tests?* 2015. Accessed on 12.26.2022. Available at: ⟨https://stackoverflow.com/questions/31434305/⟩.

STEINMACHER, I.; CONTE, T. U.; TREUDE, C.; GEROSA, M. A. Overcoming open source project entry barriers with a portal for newcomers. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (ICSE '16), p. 273–284.

STOL, K.-J.; RALPH, P.; FITZGERALD, B. Grounded theory in software engineering research: A critical review and guidelines. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (ICSE '16), p. 120–131. Available at: ⟨https://doi.org/10.1145/2884781.2884833⟩.

TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S.; YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology*, v. 125, p. 106333, 2020.

TAHIR, A.; YAMASHITA, A.; LICORISH, S.; DIETRICH, J.; COUNSELL, S. Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. New York, NY, USA: Association for Computing Machinery, 2018. (EASE'18), p. 68–78.

TAMBURRI, D. A.; KRUCHTEN, P.; LAGO, P.; VLIET, H. v. Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, Springer, v. 6, n. 1, p. 1–17, 2015.

TEMPERO, E.; GORSCHEK, T.; ANGELIS, L. Barriers to refactoring. *Communications of the ACM*, ACM New York, NY, USA, v. 60, n. 10, p. 54–61, 2017.

TERCEIRO, A.; RIOS, L. R.; CHAVEZ, C. An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In: IEEE. *2010 Brazilian Symposium on Software Engineering*. [S.l.], 2010. p. 21–29.

TIAN, F.; LIANG, P.; BABAR, M. A. How developers discuss architecture smells? an exploratory study on stack overflow. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. [S.l.: s.n.], 2019. p. 91–100.

TORCHIANO, M.; RICCA, F.; LUCIA, A. D. Empirical studies in software maintenance and evolution. In: IEEE. *2007 IEEE International Conference on Software Maintenance*. [S.l.], 2007. p. 491–494.

TRIPATHY, P.; NAIK, K. *Software evolution and maintenance: a practitioner's approach*. [S.l.]: John Wiley & Sons, 2014.

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*. [S.l.: s.n.], 2016. p. 4–15.

VIRGÍNIO, T.; MARTINS, L.; ROCHA, L.; SANTANA, R.; CRUZ, A.; COSTA, H.; MACHADO, I. Jnose: Java test smell detector. In: *34th Brazilian Symposium on Software Engineering (SBES)*. [S.l.]: ACM, 2020.

VIRGÍNIO, T.; MARTINS, L. A.; SOARES, L. R.; SANTANA, R.; COSTA, H.; MACHADO, I. An empirical study of automatically-generated tests from the perspective of test smells. In: *34th Brazilian Symposium on Software Engineering (SBES).* [S.l.]: ACM, 2020.

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. On the influence of test smells on test coverage. In: *XXXIII Brazilian Symposium on Software Engineering (SBES).* [S.l.]: ACM, 2019.

VIRGíNIO, T.; MARTINS, L. A.; SANTANA, R.; CRUZ, A.; ROCHA, L.; COSTA, H. A. X.; MACHADO, I. On the test smells detection: an empirical study on the jnose test accuracy. *Journal of Software Engineering Research and Development*, Brazilian Computer Society, v. 9, 2021.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *ACM International Conference Proceeding Series*, 05 2014.

WOHLIN, C.; RUNESON, P.; HOST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLEN, A. *Experimentation in software engineering.* [S.l.]: Springer Science & Business Media, 2012.

YIN, R. K. *Case study research and applications: Design and methods.* [S.l.]: Sage publications, 2017.

ZHOU, M.; MOCKUS, A. Developer fluency: Achieving true mastery in software projects. In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering.* [S.l.: s.n.], 2010. p. 137–146.

ZOLFAGHARI, B.; PARIZI, R.; SRIVASTAVA, G.; HAILEMARIAM, Y. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software Practice and Experience*, v. 51, 11 2020.

APPENDIX A

SURVEY AND INTERVIEW STUDIES

This appendix presents the questionnaire used to collect data in order to understand developers' and testers' perceptions of test quality practices.

A.1 SURVEY ON TEST QUALITY PRACTICES

A.1.1 Experience with the JUnit framework

A.1.1.2 Do you have professional experience with the JUnit framework?

◯ Yes

◯ No

A.1.2 Demographic information and participant experience

A.1.2.1. Which state do you live in?

Choose

A1.2.1. Education level

Choose

A.1.2.2. How long have you been working in the industry?

◯ Up to 1 year

◯ Up to 5 year

◯ Up 10 1 year

◯ Over 10 years

A1.2.3. How long have you been working with the JUnit framework?

◯ Up to 1 year

◯ Up to 5 year

◯ Up 10 1 year

◯ Over 10 years

A.1.3 Developers' perspective on creating and maintaining tests

A.1.3.1. How are test cases created?

◯ Manual
◯ Automatic
◯ Manual and automatic
◯ Other

A.1.3.2. How often are the tests maintained?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.3.3. How are test cases maintained?

◯ Manual
◯ Automatic
◯ Manual and automatic
◯ Other

A.1.3.4. Which test code generation tools are used?

☐ None

☐ EvoSuite

☐ Randoop

☐ Other

A.1.4 Developers' perspective on test quality

In this section, eight examples of test code in JUnit will be presented, followed by some questions about each of these examples individually. The participant intends to analyze the example carefully and answer the questions based on their experience and day-to-day practice.

A.1.4.1. Example of Test 01 (#EX01): Test methods that present multiple assertions without explanation. Lines 4 to 6 of the following example.

A.1.4.1.1 How often do you write test code similar to the one shown #EX01?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)

```java
public void testGetValues() throws Trowable {
    MockPartial mock = new MockPartial();
    int[] vals = mock.getValues();
    assertEquals(2, vals.length);
    assertEquals(1970, vals[0]);
    assertEquals(1, vals[1]);
    }
```

◯ Very often (≅ 100%)

A.1.4.1.2 Do you think that #EX01 can affect the maintainability of test code?

A.1.4.1.3 How often do you refactor #EX01 in test code?

◯ Never (≅ 0%)

◯ Rarely (≅ 25%)

◯ Sometimes (≅ 50%)

◯ Often (≅ 75%)

◯ Very often (≅ 100%)

A.1.4.1.4 Do you think it's necessary to refactor the #EX01 code?

A.1.4.1.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX01?

◯ None

◯ Few severe

◯ Moderately severe

◯ Severe

◯ Very severe

A.1.4.2. Test Example 02 (#EX02): Test methods that contain conditional logic statements (examples: if-else, while, for). Lines 7, 9 and 14 in the following example.

A.1.4.2.1 How often do you write test code similar to the one shown #EX02?

◯ Never (≅ 0%)

◯ Rarely (≅ 25%)

◯ Sometimes (≅ 50%)

◯ Often (≅ 75%)

◯ Very often (≅ 100%)

A.1.4.2.2 Do you think that #EX02 can affect the maintainability of test code?

A.1.4.2.3 How often do you refactor #EX02 in test code?

◯ Never (≅ 0%)

```java
@test
public void testSpinner() {
    for (Map.Entry entry : sourcesMap.entrySet()){

        String id = entry.getKey();
        Object resultObject = resultMap.get(id);
        if (resultObject instanceof EventsModel){
            EventsModel result = (EventsModel) resultObject{
                if (result.testSpinner.runTest){
                    AnswerObject answer = new AnswerObject (entry.getValue(),"", new CookieManager(),"");
                    EventScraper Scraper = new EventScraper (RuntimeEnvironment.application, answer);
                    SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                    AssertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                    for (int i =0; 1 <spinnerAdapter.getCount(); i++){
                        AssertEquals(spinnerAdapter.getitem(i), result.testSpinner.data.get(i));
                    }
                }
            }

        }
    }
    }
```

◯ Rarely (≅ 25%)

◯ Sometimes (≅ 50%)

◯ Often (≅ 75%)

◯ Very often (≅ 100%)

A.1.4.2.4 o you think it's necessary to refactor the #EX02 code?

A.1.4.2.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX02?

◯ None

◯ Few severe

◯ Moderately severe

◯ Severe

◯ Very severe

A.1.4.3. Example of Test 03 (#EX03): Test methods that check the same assertion more than once in the same method but with different values. Lines 6 and 8 in the following example.

```java
1  public void testPluralAffixParseOrder() {
2      PeriodoFormatter f = build.appendDays().appendSuffix("days", "days").toFormatter();
3      String twoDays = Period.days(2).toString(f);
4  
5      Period period = f.parsePeriod(twoDays);
6      assertEquals(Period.days(2, periodo);
7      perid = f.parseperido(twoDays.toUpperCase(Locale.ENGLISH));
8      assertEquals(Period.days(2), period);
9      }
```

A.1.4.3.1 How often do you write test code similar to the one shown #EX03?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.3.2 Do you think that #EX02 can affect the maintainability of test code?

A.1.4.3.3 How often do you refactor #EX03 in test code?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.3.4 o you think it's necessary to refactor the #EX03 code?

A.1.4.5.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX03?

◯ None
◯ Few severe
◯ Moderately severe
◯ Severe
◯ Very severe

A.1.4.4. Test Example 04 (#EX04): Test methods that test a production method that generates an exception (try/catch) and therefore use try/catch to handle the result returned by the production method. Lines 6-10 of the following example.

```java
1  @Test
2  public void testParserStringToDate() {
3      long expectedTime = 1266710400L;
4      Date date;
5
6      try{
7          date = DateUtils.paserToDate("2010-02-21", "yyyy-MM-dd");
8      } catch (PaserException e){
9          fail(e.getMessage());
10     }
11
12     assertEquals(expectedTime, date.getTime());
13 }
```

A.1.4.4.1 How often do you write test code similar to the one shown #EX04?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.4.2 Do you think that #EX04 can affect the maintainability of test code?

A.1.4.4.3 How often do you refactor #EX04 in test code?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.4.4 o you think it's necessary to refactor the #EX03 code?

A.1.4.4.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX04?

◯ None
◯ Few severe
◯ Moderately severe
◯ Severe
◯ Very severe

A.1.4.5. Test Example 05 (#EX05): Test methods that test a production method that generates an exception (try/catch) and therefore use try/catch to handle the result returned by the production method. Lines 6-10 of the following example.

```
 1  @Test
 2  public void saveImage_noImageFile_ko() throws IOExpection {
 3      File outputFile = File.createTempFile("prefix", "png", new File("/tmp"));
 4
 5      ProductImage image = new ProductImage("01010101010101", ProductImageField.FRONT, outputFile);
 6      response response = serviceWrite.saveImage(image.getCode(), image.getImguploadFront(), image.
            ↪ getImguploadIngredents(), image.getImguploadNutrition()).execute();
 7
 8      assertTrue(response.isSucess);
 9      assertThatJason(response.body()).node("status").isEqualTo("status not ok");
10  }
```

A.1.4.5.1 How often do you write test code similar to the one shown #EX05?

◯ Never (≅ 0%)

◯ Rarely ($\cong 25\%$)
◯ Sometimes ($\cong 50\%$)
◯ Often ($\cong 75\%$)
◯ Very often ($\cong 100\%$)

A.1.4.5.2 Do you think that #EX05 can affect the maintainability of test code?

A.1.4.5.3 How often do you refactor #EX05 in test code?

◯ Never ($\cong 0\%$)
◯ Rarely ($\cong 25\%$)
◯ Sometimes ($\cong 50\%$)
◯ Often ($\cong 75\%$)
◯ Very often ($\cong 100\%$)

A.1.4.5.4 o you think it's necessary to refactor the #EX05 code?

A.1.4.5.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX05?

◯ None
◯ Few severe
◯ Moderately severe
◯ Severe
◯ Very severe

A.1.4.6. Test Example 06 (#EX06): Test methods that test a production method that generates an exception (try/catch) and therefore use try/catch to handle the result returned by the production method. Lines 6-10 of the following example.

```
1  @Test
2  public void stestGetContrast() {
3      ApiService apiService = new ApiService();
4      ConsumerDataSource consumerDataSource = new ConsumerDataSource(apiService);
5      List<Contract> contract = consumerDataSource.getContracts(CLIENT_ID);
6
7      Tread.sleep(500); //wait for apiService
8      assertEquals(ExpectedContracts, contract);
9  }
```

A.1.4.6.1 How often do you write test code similar to the one shown #EX06?

◯ Never ($\cong 0\%$)
◯ Rarely ($\cong 25\%$)
◯ Sometimes ($\cong 50\%$)

◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.6.2 Do you think that #EX06 can affect the maintainability of test code?

A.1.4.6.3 How often do you refactor #EX06 in test code?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.6.4 o you think it's necessary to refactor the #EX06 code?

A.1.4.6.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX06?

◯ None
◯ Few severe
◯ Moderately severe
◯ Severe
◯ Very severe

A.1.4.7. Test Example 07 (#EX07): Test methods that test a production method that generates an exception (try/catch) and therefore use try/catch to handle the result returned by the production method. Lines 6-10 of the following example.

```
1  /* ** Test method withoud an assertion startement and non-descriptive name8** */
2  @Test
3  public void hitGetOOICategoriesApi() throws Exception {
4      POICategories poiCategories = apiClient.getPOICategories(16);
5      for (POICategory category : poiCategories){
6      System.out.println(category.name() + ":" + category);
7      }
8  }
```

A.1.4.7.1 How often do you write test code similar to the one shown #EX07?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.7.2 Do you think that #EX07 can affect the maintainability of test code?

A.1.4.7.3 How often do you refactor #EX07 in test code?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.7.4 o you think it's necessary to refactor the #EX07 code?

A.1.4.7.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX07?

◯ None
◯ Few severe
◯ Moderately severe
◯ Severe
◯ Very severe

A.1.4.8. Test Example 08 (#EX08): Test methods that test a production method that generates an exception (try/catch) and therefore use try/catch to handle the result returned by the production method. Lines 6-10 of the following example.

```
1  public class testReadImageFromCache() {
2      Image expectedImage = new Image(readBitmapfromAssets(PRODUCT_IMG);
3  
4      Image image = cache.readImage("product.png");
5  
6      assetsEquals("Read image from cache", expectedImage, image);
7  }
8  
9  @test
10 public testDeleteImageFromCache() {
11 
12     boolean deleteSucess = cache.deleteImage("product.png");
13 
14     assertTrue(deleteSucess);
15 }
```

A.1.4.8.1 How often do you write test code similar to the one shown #EX08?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)

◯ Very often (≅ 100%)

A.1.4.8.2 Do you think that #EX08 can affect the maintainability of test code?

A.1.4.8.3 How often do you refactor #EX08 in test code?

◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

A.1.4.8.4 o you think it's necessary to refactor the #EX08 code?

A.1.4.8.5 What degree of severity, in terms of comprehensibility and maintainability, do you consider for #EX08?

◯ None
◯ Few severe
◯ Moderately severe
◯ Severe
◯ Very severe

A.1.5 Developers' perspective on Test Smells

Test smells are anti-patterns in test code. They are considered bad design and implementation choices that can affect the comprehensibility and maintainability of tests.

A.1.5.1 Have you ever heard of test smells?

◯ Yes
◯ No

A.1.5.2 Which of these test smells do you know?

☐ None

☐ Assertion Roulette

☐ Conditional Test Logic

☐ Constructor Initialization

☐ Default Test

☐ Duplicate Assert

☐ Eager Test

☐ Empty Test

☐ Exception Handling

☐ For Testers Only

☐ General Fixture

☐ Ignored Test

☐ Indirect Testing

☐ Lazy Test

☐ Magic Number Test

☐ Mystery Guest

☐ Redundant Assertion

☐ Redundant Print

☐ Resource Optimism

☐ Sensitive Equality

☐ Sleepy Test

☐ Test Code Duplication T

☐ est Run War

☐ Unknown Test

A.1.6 Acknowledgement

Thank you very much for taking part in our study! Are you interested and willing to take part in our interview on the same topic? If so, please let us know your e-mail address or telephone number.

APPENDIX B

GUIDELINE EVALUATION

In this Appendix, we will present the activity we carried out to validate the guidelines.

B.1 EXPERIMENTAL DESIGN

In this study, we adopted a randomized design, where each group received a single treatment. We selected two groups of developers and classified them as Core and Peripheral. Each group consisted of 5 developers—2 core developers and 3 peripheral developers—making a total of 10 participants. We selected developers working on Java language projects and using the JUnit testing framework.

For this task, the developers were required to create a test class for a production class from the `Calculator` project without introducing test smells. For this activity, one group of developers received guidance via a guideline, while the other group did not, as shown in Tables B.1 and B.2 below:

Table B.1: Treatment with guideline

Task 1 - treatment with guideline - Calculator project	
WITH GUIDELINES	It is a set of guidelines for preventing test smells. The guideline contains examples of test methods that contain some structures that are harmful to test code and examples of how to create test cases without inserting these structures.
Calculator project	A calculator is an electronic device designed to perform mathematical operations. It is commonly used to perform fast and accurate calculations in a variety of contexts.
Scenario	The developer will receive a production class containing some test methods and will be asked to create a test class for this production class with the help of a guideline.
Task	1 - Open the IDE 2 - Import the zipped package with the name calculator 3 - Then select the SRC: 4 - Create the test class according to the requested activities.

Table B.2: Treatment without guideline

Task 1 - treatment without guideline - Calculator project	
WITHOUT GUIDELINES	No instruction
Calculator project	A calculator is an electronic device designed to perform mathematical operations. It is commonly used to perform fast and accurate calculations in a variety of contexts.
Scenario	The developer will receive a production class containing some test methods and will be asked to create a test class for this production class with the help of a guideline.
Task	1 - Open the IDE 2 - Import the zipped package with the name calculator 3 - Then select the SRC: 4 - Create the test class according to the requested activities.

B.2 STUDY ON TEST QUALITY PRACTICES

B.2.1 Informed Consent Form

B.2.1.1. I declare that I have read and understood the objectives, risks, and benefits of participating in this research.

◯ I agree

B.2.2 Experience with the JUnit framework

B.2.2.1. 1. Do you have experience with the Java programming language and the JUnit4 framework?

◯ Yes
◯ No

B.2.3 Participants' Characterization

B.2.3.1. In which state do you work?

Choose

B.2.3.2. What is your level of education?

◯ Undergraduate student
◯ Full Undergraduate student
◯ Master's degree student
◯ Full Master's degree
◯ PhD student

◯ Full PhD

B.2.3.3. Where are you currently working?

◯ Academia
◯ Industry
◯ Academia and industry

B.2.3.4. What is your role in the company?

◯ Researcher
◯ Test engineer
◯ Developer/programmer
◯ Quality analyst (QA)
◯ Others

B.2.3.5. How much experience do you have in the job (in years)?

◯ < 1 year
◯ >= 1 year and < 5 years
◯ >= 5 years and < 10 years
◯ >= 10 years

B.2.3.6. How much experience do you have with Java (in years)?

◯ < 1 year
◯ >= 1 year and < 5 years
◯ >= 5 years and < 10 years
◯ >= 10 years
◯No experience

B.2.3.7. How long have you been working with the JUnit framework (in years)?

◯ < 1 year
◯ >= 1 year and < 5 years
◯ >= 5 years and < 10 years
◯ >= 10 years
◯ No experience

B.2.3.7. How are the test cases developed?

◯ Manually
◯ Automated
◯ Manual and automated

B.2.3.8. How often do you develop test cases?
◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

B.2.3.9. How often do you make weekly commits?

B.2.3.10. Would you say that you are responsible for critical components or modules of the project?
◯ Yes
◯ No

B.2.3.11. How often do these test cases receive maintenance?
◯ Never (≅ 0%)
◯ Rarely (≅ 25%)
◯ Sometimes (≅ 50%)
◯ Often (≅ 75%)
◯ Very often (≅ 100%)

B.2.3.12. How often do these test cases receive maintenance?
◯ None
◯ EvoSuite
◯ Randoop

B.2.3.12. Do you know about test smells?
◯ Yes. I know what test smells are.
◯ No. I don't know what test smells are.
◯ I'm a researcher working on topics related to test smells.
◯ I know what it is, but I haven't worked with test smells.

◯ I've never heard of it.

B.2.3.12. Acknowledgement:
Thank you very much for your interest in joining our research!
We are currently looking for developers with knowledge of the Java language and the JUnit4 framework to take part in our study.

B.2.4 Guideline Evaluation

B.2.4.1 Tools to create test classes.

- IntelliJ IDEA
- Visual Studio Code

Note: For this research, the use of ChatGPT or similar cannot be used.

B.2.4.2 Production classes description

Figure B.1 show Production classes description.

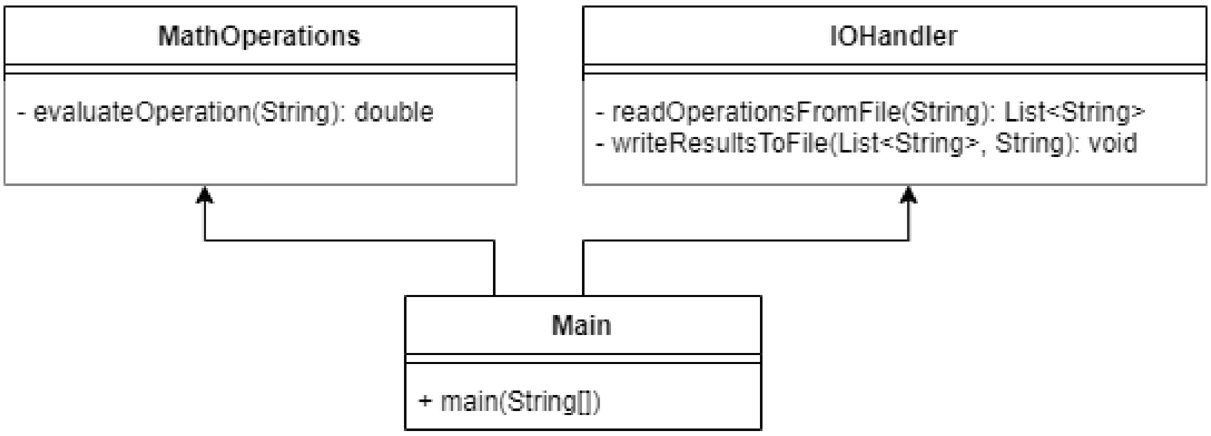

MathOperations
- evaluateOperation(String): double
IOHandler
- readOperationsFromFile(String): List<String>
- writeResultsToFile(List<String>, String): void
Main
+ main(String[])

Figure B.1: Diagram 1 - Relationship between the class MathOperations and IOHandler

1) MathOperations:

- MathOperations is the production class responsible for performing an operation between two operands.

2) IOHandler

- IOHandler reads an external file to perform several math operations, then saves the output in another file
- The expected format in the file is `operand operator operand` per line

3) Main

- It handles the calls for the MathOperations and IOHandler
- It also creates a thread pool with a fixed number of threads to perform calculations in parallel

B.2.4.3 Task #1 - Testing the edge values for calculations

Objective: Create test cases to thoroughly test the divide method of the MathOperations class to verify the edge values for the division operation.

Test steps: Considering the class MathOperations, create a test class named MathOperationsTest for the scenarios

1. Division of positive integers (dividend > divisor)
2. Division of negative integers (dividend <0, divisor> 0)
3. Division by zero (divisor == 0)

B.2.4.4 Task #2 - Reading from an External Database File

Objective: Create test cases for production classes that involve reading data from external files, such as a database.

Test steps: Considering the IOHandler class, please create a IOHandlerTest class to verify the scenarios:

1. Test the IOHandler ability to read data from the external database file
2. Verify that the content read from the file matches the expected format `operand operator operand`, e.g., `1 + 5`

B.2.4.5 Task #3 - Testing parallel calculations

Objective: Create a test case that verifies the evaluateOperation method of the Math-Operations class correctly evaluates mathematical operations in parallel and produces the expected results.

Test steps: Considering the class MathOperations, create a test class named Math-ParallelOperationsTest for the scenario:

1. Create a thread pool with a fixed number of threads
2. Create a task to execute the evaluateOperation method in parallel with the provided mathematical operation
3. Verify whether the result obtained from the completed task is correct

B.2.5 Feedback from the evaluation of the study on test quality practices

In this feedback phase, we collected information from the two groups with and without the guideline.

B.2.5.1 Did you use the guidelines to develop the test cases?

○ Yes
○ No

B.2.5.1 (1) Regarding the utilization (U) of the guidelines, rate the items from "Extremely unlikely" (1) to "Extremely likely" (6): (see Table B.3).

B.2.5.2. Regarding the ease of use (E) of the guidelines, please rate the items from "Extremely unlikely" (1) to "Extremely likely" (6): (see Table B.4).

B.2.5.3. Regarding the use (S) of the guidelines in the future, rate the items from "Extremely unlikely" (1) to "Extremely likely" (6): (see Table B.5).

Table B.3: Utilization (U) of the guidelines

Utilization (U)	**1**	**2**	**3**	**4**	**5**	**6**
U1. By using the guidelines in my work, I would be able to create tests faster and with good programming practices	○	○	○	○	○	○
U2. Using the guidelines would improve my performance when creating test cases.	○	○	○	○	○	○
U3. Using the guidelines would increase my effectiveness in creating test cases.	○	○	○	○	○	○
U4. Using guidelines would make it easier to create test cases.	○	○	○	○	○	○
U5. I would find the guidelines useful for creating test cases.	○	○	○	○	○	○
U6. The guidelines used in the experiment are easy to understand and apply.	○	○	○	○	○	○
U7. Guidelines can help improve the readability and comprehensibility of tests.	○	○	○	○	○	○

Table B.4: Use (E) of the guidelines

Use (E) of the guidelines	**1**	**2**	**3**	**4**	**5**	**6**
E1. The guidelines would be easy to use.	○	○	○	○	○	○
E2. The use of the guidelines is clear and understandable.	○	○	○	○	○	○
E3. The use of guidelines is applicable to different types of projects and development contexts.	○	○	○	○	○	○
E4. It's easy to create test cases using the guidelines.	○	○	○	○	○	○

Table B.5: Use (S) of the guidelines

Ese (S) of the guidelines	**1**	**2**	**3**	**4**	**5**	**6**
S1. Assuming that the guidelines are available in my work, I anticipate that I will use it regularly in the future.	○	○	○	○	○	○
S2. I'd rather use the guidelines than not.	○	○	○	○	○	○

B.2.5.4. What are the main **challenges** you faced when applying the prevention guidelines in the project?

B.2.5.5. In your opinion, what are the **advantages** of using guidelines?

B.2.5.6. In your opinion, what are the **disadvantages** of using guidelines?

B.2.5.7. What were the **main obstacles** you encountered when developing the test cases for the calculator project?

B.2.5.8. Suppose you had a guide to help you create test cases without using bad programming practices. Would you use it to create test cases?

◯ Yes
◯ No

B.2.5.9. Acknowledgements

The research team would like to thank you for your availability. Your contribution will undoubtedly be of great value to our research. If you would like to follow the next steps in this study, please provide your e-mail address.