

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 evolution is inevitable if the systems are planned to survive in the long-term. Equally, well-understood is the necessity of having a good test suite available to ensure the quality of the current state of the software system and to ease future changes. This is especially true in the context of reusable systems since they are planned to attend for a long time a specific market niche and need to support a large number of configuration options. However, developing and maintaining a test suite is time-intensive and costly. This situation is challenging for the projects: on the one hand, tests are essential for the success of software; on the other hand, tests become a severe burden during maintenance. Even though a substantial body of literature has studied testing in reusable environments, test evolution analysis has not been addressed. In general, researchers have looked into analyzing test strategies, dynamic test selection techniques, and co-evolution of tests along with other systems artifacts. This thesis intends to improve the test evolution body of knowledge in reusable systems, investigating characteristics that indicate the effort to develop and maintain the test suite and unveiling how the reusable aspects affect the tests. The set of evidence can help researchers and practitioners to better planning the test development and evolution. This way, we employed a multi-method approach to develop the understanding of test evolution in configurable systems and unveil evidence on the topic from various sources. In the first phase of the research program, we provided an overview of the existing research related to this thesis' s subjects and presented related work to our investigation. The second phase was composed of four empirical studies. First, we performed a case study to analyze the test evolution of a large configurable system. Next, we performed a comparative study to evaluate the test evolution in 18 open-source projects from various sizes and domains in configurable systems and their similarities and differences to 18 Single Systems projects. Third, we conducted an extended study to analyze the test suite evolution in another category of reusable systems to verify whether some observations are recurring and gather new data that support the findings. Finally, we surveyed test contributors to investigate the test evolution from the development point of view and improve the findings in previous stages. This work collected a set of findings of test evolution, and these findings were strengthened by using different research methods. Our work provided a better understanding of test evolution in configurable systems by documenting evidence observed in open-source projects and test contributors. Moreover, in this Thesis, we synthesized the gathered evidence and identified open issues in this research topic. These findings are an important step to establish guidelines for addressing test evolution in configurable systems.

Understanding Test Evolution: from Highly-Configurable Systems to Software Ecosystems

Jonatas Ferreira Bastos

Tese de Doutorado

Universidade Federal da Bahia

Programa de Pós-Graduação em
Ciência da Computação

Maio | 2021

DSC | 018 | 2021

Understanding Test Evolution: from Highly-Configurable Systems to Software Ecosystems

Jonatas Ferreira Bastos

UFBA





Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

**UNDERSTANDING TEST EVOLUTION:
FROM HIGHLY-CONFIGURABLE SYSTEMS
TO SOFTWARE ECOSYSTEMS**

Jonatas Ferreira Bastos

TESE DE DOUTORADO

Salvador, Bahia – Brasil
May, 2021

JONATAS FERREIRA BASTOS

**UNDERSTANDING TEST EVOLUTION: FROM
HIGHLY-CONFIGURABLE SYSTEMS TO SOFTWARE
ECOSYSTEMS**

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. Eduardo Santana de Almeida

Salvador, Bahia – Brasil

May, 2021

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

B327 Bastos, Jonatas Ferreira.

Understanding Test Evolution: from Highly-Configurable
Systems to Software Ecosystems / Jonatas Ferreira Bastos –
Salvador, 2021.

141 f.

Orientador: Prof. Dr. Eduardo Santana de Almeida

Tese (Doutorado) – Universidade Federal da Bahia.
Instituto de Computação, 2021.

1. Software. 2. Software Engineering. 3. Ecosystem. I.
Almeida, Eduardo Santana de. II. Universidade Federal da
Bahia. III. Título.

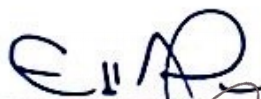
CDU 004.8

JONATAS FERREIRA BASTOS

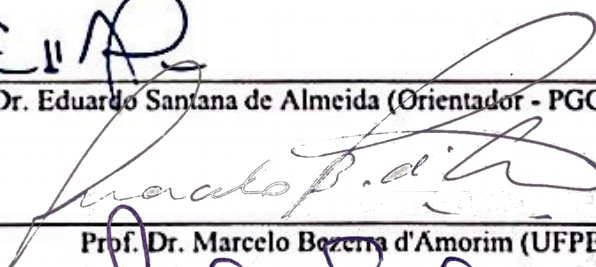
**“UNDERSTANDING TEST EVOLUTION: FROM HIGHLY-
CONFIGURABLE SYSTEMS TO ECOSYSTEMS”**

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.

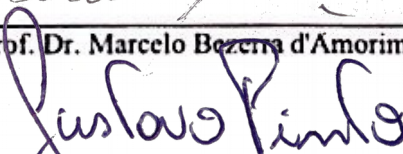
Salvador, 04 de maio de 2021



Prof. Dr. Eduardo Santana de Almeida (Orientador - PGCOMP/UFBA)




Prof. Dr. Marcelo Bezerra d'Amorim (UFPE)



Prof. Dr. Gustavo Henrique Lima Pinto (UFPA)



Profa. Dra. Christina von Flach Garcia Chavez (UFBA)



Prof. Dr. Rodrigo Rocha Gomes e Souza (UFBA)

ACKNOWLEDGEMENTS

Eu gostaria de agradecer ao meu orientador, o professor Eduardo Almeida. Esa obrigado por todo suporte durante esses anos, pelas conversas e reuniões, por acreditar e confiar em mim. Sua orientação é um dos maiores ensinamentos que levo desses anos, e espero poder repassar teus ensinamentos a meus orientados na carreira acadêmica. Gostaria de agradecer também ao professor Paulo Cesar Masieiro. Muito obrigado por todas as nossas conversas, por ser sempre tão gentil e pela disponibilidade. Sempre após as nossas reuniões, eu me sentia muito melhor. Obrigado de verdade. Eu também preciso falar do melhor grupo de pesquisa e laboratório do mundo, RiSE Labs e Lab INES! A companhia diária, o ambiente descontraído, as risadas e todos os momentos com vocês serviram como combustível para que eu pudesse seguir em frente. Magno, Glaúcia, Leandro, Iuri, Michele, Larissa, Paulo, Renata, Tassio, Alberto, Crescêncio, Jaziel, Tiago, Rose, Carla, etc. Obrigado.

Durante o doutorado, eu tive a oportunidade de ser recebido pelo professor Ahmed E. Hassan no SAIL Lab na Quenns University, Kingston, Canadá. Eu não tenho palavras suficientes para agradecer todo o suporte e incríveis conversas que eu tive durante o ano que passei em Kingston. SAIL foi um divisor de águas na minha vida. Não sou a mesma pessoa de antes. Eu aprendi muito, amadureci mais ainda, tanto profissionalmente quanto como pessoa. A maneira que faço pesquisa mudou. Minha forma de pensar também mudou e sou muito grato por tudo isso.

Também gostaria de agradecer a Gustavo Ansaldi e Iftekhar Ahmed pela grande colaboração em minha pesquisa durante o ano no Canadá. Ainda assim, gostaria de agradecer ao grupo de pesquisa SAIL, principalmente aquele que se tornou meu amigo, Filipe Cogô, pelas infinitas conversas, apoio nos momentos difíceis, riso e amizade. Por último, e não menos importante, gostaria de agradecer a todos os meus amigos em Kingston, em especial a Pedro, Plínio, João, Bruna, Delano, Lilly e Lebron. Sem eles, minha vida teria sido muito mais solitária e triste. Eles foram minha família por 1 ano e trouxeram companheirismo e alegria para minha vida. Muitos dos momentos mais felizes que tive foram por causa de vocês. Obrigado pessoal.

Um muito obrigado também a galera do ap. 203, eu não poderia ter tido melhor ambiente em momentos tão difíceis. Um muito obrigado também aos meus pais, por estarem sempre ao meu lado e me darem todo o suporte necessário e todo o amor desse mundo. Preciso também agradecer a Leninha por todo o suporte e por cuidar tão bem durante a minha ausência daquela que é minha melhor parte, Ingrid Gabirele. Ingrid minha filha, nada nesse mundo poderá devolver esse ano longe de você, e apesar de lhe dizer pessoalmente sempre, quero deixar registrado aqui, o meu mais profundo amor e admiração pelo ser humano que você tem se tornado, e por ser calma quando a vida se fez turbulenta.

Finalmente, preciso agradecer a minha irmã, sobrinhos (Iago, Jair Jr. e Joaquim). Obrigada por acreditarem em mim quando nem eu mesmo não mais acreditava. Amo muito vocês.

RESUMO

A evolução do software é inevitável se os sistemas forem planejados para sobreviver a longo prazo. Igualmente, é a necessidade de ter um bom conjunto de testes disponível para garantir a qualidade do estado atual do sistema de software e facilitar mudanças futuras. Isso é especialmente verdade no contexto de sistemas reutilizáveis, uma vez que são planejados para atender por muito tempo um nicho de mercado específico e precisam suportar um grande número de opções de configuração. No entanto, desenvolver e manter um conjunto de testes é demorado e caro. Essa situação é desafiadora para os projetos: por um lado, os testes são essenciais para o sucesso do software; por outro lado, os testes tornam-se um fardo severo durante a manutenção.

Embora um corpo substancial de literatura tenha estudado testes em ambientes reutilizáveis, a análise da evolução do teste não foi abordada. Em geral, os pesquisadores analisaram estratégias de teste, técnicas de seleção de teste dinâmico e co-evolução de testes junto com outros artefatos de sistemas. Esta tese pretende aprimorar o conhecimento da evolução de testes em sistemas reutilizáveis, investigando características que indicam o esforço para desenvolver e manter a suíte de testes e desvendando como os aspectos reutilizáveis afetam os testes. O conjunto de evidências pode ajudar pesquisadores e profissionais a planejar melhor o desenvolvimento e a evolução do teste.

Dessa forma, empregamos uma abordagem multi-métodos para desenvolver o entendimento da evolução de testes em sistemas configuráveis e desvendar evidências sobre o tema a partir de diversas fontes. Na primeira fase do programa de pesquisa, fornecemos uma visão geral da pesquisa existente relacionada aos assuntos desta tese e apresentamos trabalhos relacionados à nossa investigação. A segunda fase foi composta por quatro estudos empíricos. Primeiramente, realizamos um estudo de caso para analisar a evolução do teste de um grande sistema configurável. Em seguida, realizamos um estudo comparativo para avaliar a evolução do teste em 18 sistemas configuráveis e open source, de vários tamanhos e domínios, investigando assim as suas semelhanças e diferenças com 18 projetos não configuráveis. Terceiro, realizamos um estudo estendido para analisar a evolução do conjunto de testes em outra categoria de sistemas reutilizáveis verificando assim se algumas observações são recorrentes ao mesmo tempo que coletamos novos dados que suportam as descobertas. Por fim, pesquisamos os colaboradores do teste para investigar a evolução do teste do ponto de vista do desenvolvimento e melhorar as descobertas nos estágios anteriores.

Este trabalho coletou um conjunto de achados da evolução dos testes, e esses achados foram fortalecidos por meio de diferentes métodos de pesquisa. Nosso trabalho forneceu uma melhor compreensão da evolução do teste em sistemas configuráveis documentando evidências observadas em projetos de código aberto e colaboradores de teste. Além disso, nesta Tese, sintetizamos as evidências coletadas e identificamos questões em aberto neste

tópico de pesquisa. Essas descobertas são um passo importante para estabelecer diretrizes para abordar a evolução do teste em sistemas configuráveis.

Palavras-chave: Teste; Evolução; Sistemas altamente configuráveis; Variabilidade; Estudo Empírico;

ABSTRACT

Software evolution is inevitable if the systems are planned to survive in the long-term. Equally, well-understood is the necessity of having a good test suite available to ensure the quality of the current state of the software system and to ease future changes. This is especially true in the context of reusable systems since they are planned to attend for a long time a specific market niche and need to support a large number of configuration options. However, developing and maintaining a test suite is time-intensive and costly. This situation is challenging for the projects: on the one hand, tests are essential for the success of software; on the other hand, tests become a severe burden during maintenance.

Even though a substantial body of literature has studied testing in reusable environments, test evolution analysis has not been addressed. In general, researchers have looked into analyzing test strategies, dynamic test selection techniques, and co-evolution of tests along with other systems artifacts. This thesis intends to improve the test evolution body of knowledge in reusable systems, investigating characteristics that indicate the effort to develop and maintain the test suite and unveiling how the reusable aspects affect the tests. The set of evidence can help researchers and practitioners to better planning the test development and evolution.

This way, we employed a multi-method approach to develop the understanding of test evolution in configurable systems and unveil evidence on the topic from various sources. In the first phase of the research program, we provided an overview of the existing research related to this thesis's subjects and presented related work to our investigation. The second phase was composed of four empirical studies. First, we performed a case study to analyze the test evolution of a large configurable system. Next, we performed a comparative study to evaluate the test evolution in 18 open-source projects from various sizes and domains in configurable systems and their similarities and differences to 18 Single Systems (SS) projects. Third, we conducted an extended study to analyze the test suite evolution in another category of reusable systems to verify whether some observations are recurring and gather new data that support the findings. Finally, we surveyed test contributors to investigate the test evolution from the development point of view and improve the findings in previous stages.

This work collected a set of findings of test evolution, and these findings were strengthened by using different research methods. Our work provided a better understanding of test evolution in configurable systems by documenting evidence observed in open-source projects and test contributors. Moreover, in this Thesis, we synthesized the gathered evidence and identified open issues in this research topic. These findings are an important step to establish guidelines for addressing test evolution in configurable systems.

Keywords: Test; Evolution; Highly-Configurable Systems; Software Ecosystem; Vari-

ability; Empirical Study.

CONTENTS

List of Figures	xiv
List of Tables	xvii
List of Acronyms	xvii

I Overview

Chapter 1—Introduction	3
1.1 Motivation	3
1.2 Objective	4
1.3 Research Questions (RQs)	4
1.4 Research Methodology	5
1.4.1 Research Design	5
1.5 Contributions	6
1.6 Out of Scope	8
1.7 Organization of the Thesis	9

II Background

Chapter 2—Main Concepts and Foundations	13
2.1 Software Reuse	13
2.2 Reusable Systems	14
2.2.1 Highly-Configurable Systems (HCS)	14
2.2.1.1 Variability Implementation in HCS.	14
2.2.2 Ecosystems	15
2.3 Software Testing	15
2.3.1 Testing Levels	16
2.4 Chapter Summary	17

Chapter 3—An Overview of the State-of-the-art and Related Work	19
3.1 Connection between test, reusable systems, and evolution	19
3.2 Studies on test evolution	20
3.2.1 Testing in Configurable Systems	21
3.2.2 Testing in Ecosystems	22
3.2.3 Studies Comparison	23
3.3 Chapter Summary	23

III Empirical studies

Chapter 4—Test Suite Evolution in a Highly-Configurable System: A Case Study on the Linux Test Project (LTP)	27
4.1 Background	28
4.1.1 The Linux Test Project (LTP)	28
4.1.2 LTP Structure	28
4.1.3 Basic test structure	29
4.2 Methodology	30
4.2.1 GQM Model	30
4.2.2 Research Questions	31
4.2.3 Data Collection	32
4.3 Results	32
4.3.1 RQ1 - What is the effort to develop the test suite?	32
4.3.1.1 M1 - Test Cases:	33
4.3.1.2 M2 - Unit Size:	34
4.3.1.3 M3 - Contributors:	36
4.3.2 RQ2 - What is the maintainability of the test suite?	38
4.3.2.1 M2 - Unit Size:	38
4.3.2.2 M4 - Unit Complexity:	39
4.3.2.3 M5 - Duplication:	41
4.3.2.4 M6 - Dependence:	43
4.3.3 RQ3 - How does the test suite change?	44
4.3.3.1 M7 - Test Case Changes:	44
4.3.3.2 M8 - Test Program Changes:	46
4.3.4 RQ4 - How does variability affect the maintainability of the test suite?	48
4.3.4.1 M9 - Lines of Feature Code (LOF):	48
4.3.4.2 M10 - Number of Feature Constants (NOFC):	50
4.3.4.3 M11 - Scattering Degree (SD):	52
4.3.4.4 M12 - Tangling Degree (TD):	54
4.4 Discussion	56
4.5 Threats to Validity	58
4.5.1 External Validity	58

4.5.2	Construct Validity	59
4.6	Chapter Summary	59
Chapter 5—Test Evolution in Configurable Systems and Single Systems: A comparative Study		61
5.1	Methodology	61
5.1.1	Study Subjects	62
5.1.2	Data Collection	62
5.1.3	Data Preparation	63
5.1.4	Tools Selection	63
5.2	Results	66
5.2.1	RQ1- How much effort is required to evolve test suite?	66
5.2.1.1	Number of Contributors:	66
5.2.1.2	Man-Month (M/M):	67
5.2.1.3	Modified files:	67
5.2.1.4	Assertions:	68
5.2.2	RQ2 - How maintainable is the test suite and how that evolves? .	71
5.2.2.1	Unit Size:	71
5.2.2.2	Unit Complexity:	72
5.2.2.3	Unit Dependence:	73
5.2.2.4	Duplication:	74
5.3	Discussion	76
5.4	Threats to Validity	77
5.5	Chapter Summary	78
Chapter 6—Test Evolution in a Software Ecosystem: The extended study on npm packages		79
6.1	Data Collection	80
6.2	Results	81
6.2.1	RQ1: How often do packages perform testing?	81
6.2.2	RQ2: How does test code evolve?	84
6.2.3	RQ3: How is the ownership of test code?	88
6.3	Discussion	92
6.4	Threats to Validity	94
6.5	Chapter Summary	95
Chapter 7—Survey with Test Contributors to Understand the Test Evolution		97
7.1	Related Work	97
7.2	Methodology	98
7.3	Results	99
7.3.1	General Information	100
7.3.2	Effort Characteristics	100

7.3.3	Maintainability sub-characteristics	104
7.3.4	Main Challenges	106
7.4	Discussion	106
7.5	Threats to Validity	107
7.6	Chapter Summary	107

IV Conclusions

Chapter 8—Research Synthesis and Evaluation of the Multi-Method Approach 111

8.1	Justification for the Multi-Method Approach	112
8.2	Research Synthesis	112
8.3	Summary of the Findings	113
8.4	Findings of the Multi-Method Approach	119
8.5	Research Question Analysis	120
8.6	Multi-Method Approach Evaluation	121
8.7	Chapter Summary	121

Chapter 9—Concluding Remarks and Future Work 123

9.1	Summary of Contributions	124
9.2	Future Work	125
9.3	Concluding Remarks	125

LIST OF FIGURES

1.1	Research Design.	7
1.2	Schematic overview of the thesis structure.	10
4.1	LTP Organization.	29
4.2	GQM model of the study.	31
4.3	Data collection process.	32
4.4	Number of Test Cases per Release.	33
4.5	Test Programs evolution per subsystem.	34
4.6	Quantity of Test Suites and avg. of test cases in each one.	34
4.7	LOC per releases.	35
4.8	LOC in each subsystem per releases.	35
4.9	Number of Contributors per month.	36
4.10	LOC per man-month (log10).	37
4.11	The productivity per Man-Month.	37
4.12	Percentiles of Test Program (LOC).	38
4.13	Accumulated Cyclomatic Complexity per Releases.	40
4.14	Average of Cyclomatic Complexity per Function.	40
4.15	Average of Cyclomatic Complexity per Subsystem.	40
4.16	Percentiles of unit complexity per Function.	41
4.17	Percentage of Total Cloned Functions.	42
4.18	Percentage of File Associated with Clones.	42
4.19	Average of Unit Dependence per Release.	43
4.20	Percentiles of Unit Dependence.	44
4.21	LPT evolution from the test cases change point-of-view.	45
4.22	Changes in the number of Test Cases between releases.	45
4.23	Number of Releases that a Test Case Remains in LTP.	46
4.24	Number of changes in code files per releases.	47
4.25	Test programs changed per subsystem.	47
4.26	Test programs with more changes.	48
4.27	Lines of Feature Code evolution.	48
4.28	LOF in each subsystem.	49
4.29	Fraction of cpp-annotated code (LOF/LOC) per Release.	50
4.30	Number of Features Constants (NOFC).	50
4.31	NOFC in each subsystem.	51
4.32	Highest NOFC values per Release.	51
4.33	SD Evolution.	52
4.34	Percentiles of SD per Release.	53

4.35	Evolution of the <code>_sparc_</code> feature constant.	53
4.36	Tangling Degree Evolution.	54
4.37	Percentiles of TD per Release.	55
4.38	Feature expressions with high TD evolution.	55
5.1	Overview of our subjects selection and data collection approach.	62
5.2	Contributors working in test files over time.	67
5.3	Man-Month productivity in test files per month.	68
5.4	Modified test files per Contributors.	69
5.5	Distribution of Modified Files.	69
5.6	Average of assertions in Test Files.	70
5.7	Assertions density in Test Files.	70
5.8	Unit size over time.	72
5.9	Average of Cyclomatic Complexity in test files.	73
5.10	Average of Unit Dependence of the test files.	74
5.11	Percentage of Total Cloned Functions.	75
6.1	Overview of our data collection approach.	80
6.2	Proportion of popular packages that perform or not tests.	82
6.3	The number of releases taken for a package to start having tests.	83
6.4	Test framework adoption distribution by popular packages.	83
6.5	Proportion of Downgrades and Updates for releases with and without test.	84
6.6	Example of the division of a packages' time-frame.	85
6.7	Percentage of test files.	86
6.8	Percentage of <i>tLOC</i>	86
6.9	Percentage of <i>tLOC</i>	87
6.10	Proportion of modified test files during each time-frame.	87
6.11	Average of <i>tLOC</i> changed per test file.	88
6.12	Percentage of test contributors.	90
6.13	Average of modified <i>tLOC</i> by contributors.	90
6.14	Distribution of modified test files.	91
6.15	One-time contributor (OTC).	91
6.16	Percent of contributors that perform changes in the test files.	92
7.1	Survey Steps.	98
7.2	Contributors Information.	100
7.3	Educational Qualification.	100
7.4	Contributors Experience.	101
7.5	The necessity of more contributors at the beginning of a project.	101
7.6	The negative impact of high workload.	102
7.7	Contributors only involved in test activities.	103
7.8	Concentration of work.	103
7.9	Code Complexity.	104
7.10	Code Dependence.	105
7.11	Code Clone.	105

LIST OF TABLES

1.1	Publications during the Ph.D. research.	8
3.1	Comparison of the characteristics evaluated in our study and the related works.	23
4.1	Summary of Findings.	56
5.1	HCS Projects Selected.	64
5.2	SS Projects Selected.	65
5.3	Wilcoxon Rank Sum test (p -value) and Cliff's Delta (d) for SS vs. HCS projects.	66
5.4	Thresholds and Values for Unit Level Metrics.	72
5.5	Percentage cloned LOC by level of Similarity.	75
6.1	File type classifications examples.	81
8.1	Summary of Findings.	114

LIST OF ACRONYMS

HCS	Highly-Configurable Systems
SS	Single Systems
SPL	Software Product Lines

PART I

OVERVIEW

INTRODUCTION

Lehman was one of the pioneers to say that a software system must evolve, or it becomes progressively less satisfactory [1, 2]. We also know that due to ever-changing surroundings, new business needs, new regulations, and also due to the people working with the system, the software is in a semi-permanent state of flux [3]. Additionally, the increasing lifespan of most software systems [4] leads to a situation where an even higher fraction of the total budget of a project is spent during the maintenance or evolution phase, considerably outweighing the initial development costs of a system [5].

Software, however, is multidimensional, and so is the development process behind it. This multidimensionality lies in the fact that other artifacts are needed to develop high-quality source code, and one of these artifacts is the tests [6]. According to Moonen et al. [5], tests are crucial during evolution since they are responsible for quality assurance, documentation, and confidence. However, developing and maintaining a test suite is time-intensive [5, 7], and costly [8]. According to Brooks [9], the total time devoted to testing is 50% of the total allocated time, while Kung et al. [8] suggest that 40 to 80% of the development costs of software development is spent in the testing phase. These percentages can increase even more in reusable systems since in these systems, the variability aspects, such as variation points (elements to manage variability and to facilitate the derivation of different configurations) in Highly-Configurable Systems (HCS) and dependence management in ecosystems, increase the challenges in the development of tests [10]. All these challenges put projects in a difficult situation: on the one hand, tests are essential for the success of software; on the other hand, tests become a severe burden during maintenance [11].

1.1 MOTIVATION

Due to the critical role of tests in ensuring software quality, researchers have investigated test evolution in different contexts [11, 12, 13, 14, 15]. They have looked into analyzing test strategies [10], dynamic test selection techniques [16, 17, 18], and co-evolution of tests along with other systems artifacts [19, 20]. The lack of a thorough understanding

of how test evolves in reusable systems is directly reflected in the quality of existing tools, and methodologies [5, 21, 10]. Nevertheless, no specific research has been carried out in the context of reusable systems investigating the test suite evolution on two main aspects: *(i)* Development effort and *(ii)* Maintainability. These aspects are tremendously impacted by the reusability since an artifact can be shared among several products in this kind of environment, and any change in this asset may affect all related products, making the test evolution more difficult [10].

Thus, the investigation of software test evolution is essential. A better understanding of this evolution can allow us, in the medium-term, to come up with prediction models, guidelines, and best practices that will enable the community to improve their current practices, tools and to make the test process less expensive and time-consuming.

1.2 OBJECTIVE

In this work, the main goal is to advance the test evolution body of knowledge in reusable systems, evaluating characteristics that indicate the effort to develop and maintain the test suite, and identifying reusable aspects that can affect the tests.

This research has the following specific objectives:

Research Goal 1: Contribute to a better understanding test evolution focusing on two main aspects: *(i)* Development effort, and *(ii)* Maintainability.

Research Goal 2: Provide empirical evidence about test evolution by analyzing open-source projects from various domains and different sizes in reusable environments.

1.3 RESEARCH QUESTIONS (RQS)

On the basis of such defined goals, we established the following research questions that drive this investigation:

RQ1 - What is the effort required to evolve the test suite? Projects evolve, among other reasons, due to bug fixes, new features, and refactoring. These changes can be followed by changes in the test code [22]. The evaluation of the performed changes and the stability of such tests can reveal valuable information about the effort needed to evolve the test suite. [5, 23]. Understanding how tests change over time is important for development teams to allocate personnel and resources to test tasks effectively and reduce the test overhead in regular development tasks, such as fixing defects and adding new tests.

RQ2 - How maintainable is the test suite? Test code has similar requirements for maintenance as production code [23], and it is essential to ensure that it is clear to read, understand, and make changes. In the same way, integrating the execution of the tests in the development process requires that the tests run efficiently. Thus, the monitoring and detection of possible points of low quality in the test code are crucial. Additionally, understanding how the test suites are maintained can form the base to investigate the interactions between tests and the system under evolution [5].

RQ3 - How is the ownership of test code? Ownership describes whether one or more contributors have responsibility for a portion of the software system, being an important aspect to the quality and maintenance of open-source software packages [24]. Additionally, the development of open-source software projects requires balancing the workload between groups of participants [25]. As a project grows, it is unclear whether the arrangements that previously made the project work will continue to be relevant or whether new methods will be needed. The analysis of ownership concentration can provide the community with an overview of the “sustainability” of the projects, and specifically, how sustainable the tests are as the projects evolve.

RQ4 - Are there reusable aspects that affect the tests? Variability represents an essential role in software development in reusable systems [10]. However, the variability aspects have been investigated as they occur in the system code [19]. Since the test code has a different purpose, it is not clear if the test code is also affected by the variability. Thus, evaluating the variability in the tests can contribute to the development or adaptation of tools and guidelines for the test evolution in configurable systems.

1.4 RESEARCH METHODOLOGY

There has been a growing interest in empirical research in Software Engineering [26]. One of the main reasons is that the analytical research paradigm is insufficient for investigating complex real-life issues involving humans and their interactions with technology [27]. Test evolution in reusable systems is not different. Some studies investigated the test evolution of systems that adopt software reuse [19, 11, 12]. However, the reports represent single-shot studies *i.e.* only one study in the topic of investigation. According to Wood et al. [27], “one experimental study on a topic, no matter how good, cannot, in isolation, demonstrate much of anything conclusively”. Replication helps to address some of the weaknesses of single-shot studies. Still, replications typically involve repeated investigation using the same research method and are therefore vulnerable to the inherent weaknesses of any particular method [28].

Thus, we adopted a multi-method approach to have a deep understanding of test evolution in reusable systems. A multi-method approach [29], or triangulation [30] combines different, but complementary, studies. It is argued that a multi-method approach can help to address the perceived weakness of single-shot studies by attacking research problems “with an arsenal of methods that have non-overlapping weaknesses in addition to their complementary strengths” [29].

1.4.1 Research Design

We have observed among other issues, the lack of research into test evolution in reusable systems combining evidence from different sources. In addition, we have noticed that research in test evolution were centered on technological solutions that supported or partly automated the test evolution process [11, 14, 13, 31] and are based largely on anecdotal rather than scientific evidence.

While such a large gap between research and practice presents many opportunities,

it also brings many potential pitfalls. Without a mature body of theoretically founded knowledge to use as a guiding light, systematic research becomes somewhat more complicated. It is recommended that in instances where the study is broad, exploratory, and where limited research currently exists, the researcher must analyze the research project into a set of clearly defined steps [32]. On the other hand, to determine the research steps, the research methods must be selected. However, no single research method is universally applicable, and “all research approaches may have something to offer” [33]. There is a considerable range of research methods available [34], all of which have distinct strengths and weaknesses. To compensate for these weaknesses, Franz et al. [35] recommend a multi-method research design. Multi-method design is “the conduct of two or more research studies, each conducted rigorously and complete in itself, in one project” [36]. By triangulating between studies and data, more plausible interpretations can emerge.

Thus, our multi-method research design was influenced by [37], which focuses on combining research methods to both gain further understanding of the research problem, and to enrich our conclusions, as shown in Figure 1.2:

Background. The first part presents an overview of the basic concepts that guide this thesis, such as software reuse, HCS, ecosystems, and test evolution. Also, it encompasses an overview of the state-of-art of field and related work.

Empirical Studies. The second part represents the core of this investigation and comprises empirical studies to understand the test evolution in reusable systems. This part is divided into four studies: *(i)* we performed a case study to analyze the test evolution of a large configurable system; *(ii)* we conducted a comparative study to evaluate the test evolution in 18 open-source projects from various size and domains in configurable systems and its similarities and differences to 18 SS projects. The comparison allows us to identify similarities that can help to adopt techniques from SS to configurable systems and vice versa, and the differences can help us to design new strategies addressing the discrepancies; *(iii)* we performed an extended study to analyze the test suite evolution in another category of reusable systems to verify whether some observations are recurring and gather new data that support the findings; *(iv)* we surveyed test contributors to investigate the test evolution from the development point of view and improve the findings in previous stages.

Triangulation. The latter task was backed up by data obtained from the preceding task. It is the primary goal of this investigation and mainly consists of summarizing, integrating, combining, and comparing the findings, providing empirical observations and implications for the community.

1.5 CONTRIBUTIONS

Following our goals, the main contributions of this work are related to test evolution in reuse environments, and they are listed as follows:

1. *Test Evolution state-of-the-art* since we contributed to update the body of knowledge of test evolution in reusable systems;

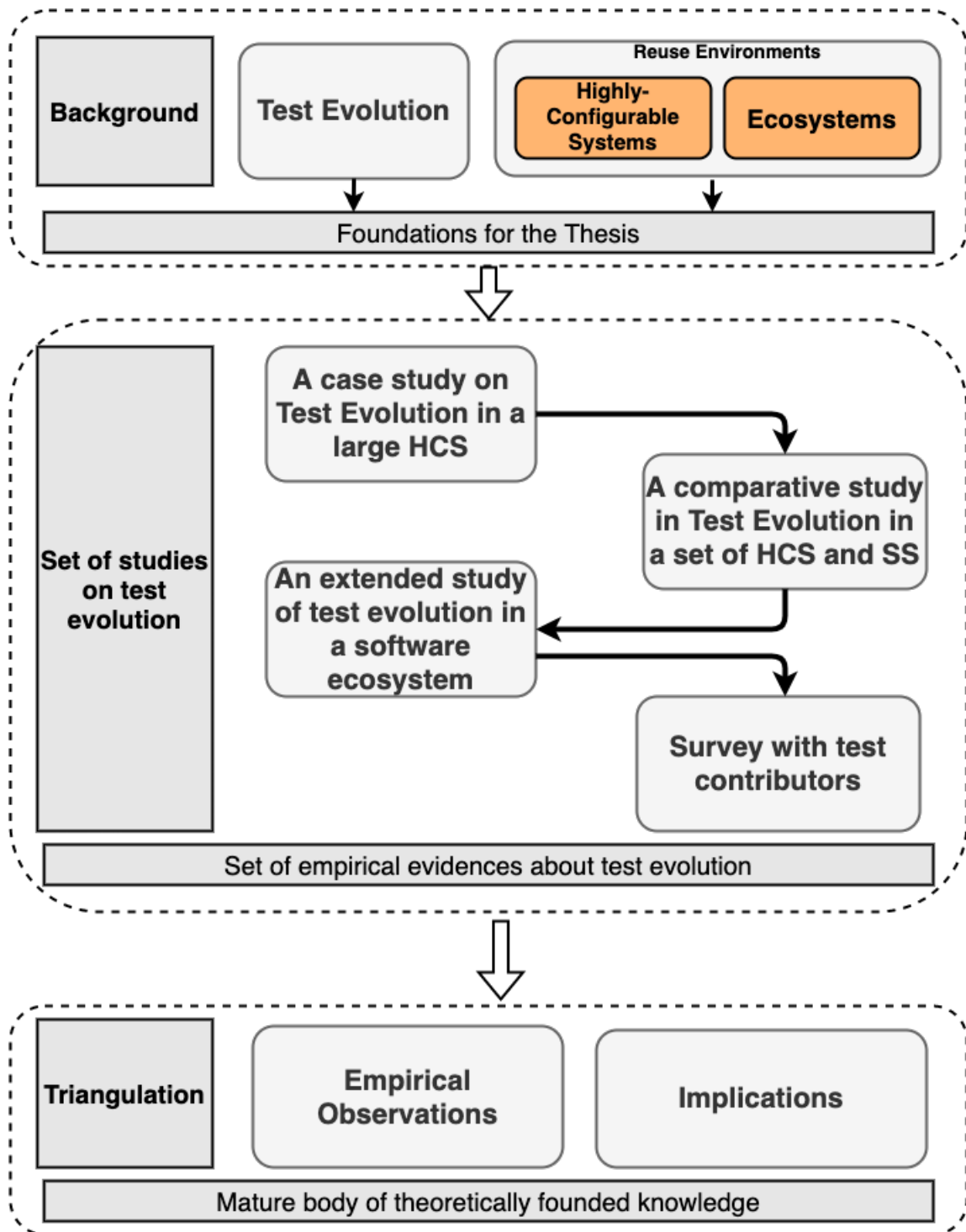


Figure 1.1: Research Design.

2. *Empirical studies on Test Evolution*, which were performed in different types of systems from various sizes and domains;
3. *Three large dataset* on test evolution. The dataset can provide researchers with a testbed that can be used for empirical research in software engineering;
4. *A set of observations* about test evolution providing static analysis of the tests code. The reports include discussions, comments, and suggestions to guide future investigations in different dimensions of test case evolution;

Based on the empirical studies defined in this thesis, we have three journal papers in evaluation, as can be seen in Table 1.1. Additionally, we have two other publications.

Table 1.1: Publications during the Ph.D. research.

Paper Title	Venue	Year
Thesis related publications		
1. Test Suite Evolution - A Study on the Linux TestProject (LTP)	JSS [under evaluation]	2021
2. Understanding Test Evolution in Highly-Configurable Systems and Single Systems from Effort and Maintainability perspectives	IST [under evaluation]	2021
3. Testing evolution in the npm ecosystem	EMSE [under evaluation]	2021
Other publications		
4. Software Product Lines Adoption: An Industrial Case Study [38]	CESI	2015
5. Software product lines adoption in small organizations [39]	JSS	2017

1.6 OUT OF SCOPE

It is essential to define the scope of this thesis. Thus, we consider as out of the scope the following topics:

- **Systematic Process.** It is known that a well-defined test evolution process can bring benefits for the organizations that adopt it. However, currently, several organizations have their software development processes and the context in which the roles and the activities are defined and coordinated, at least, in theory. Thus, this research provides a set of evidence and recommendations for organizations during the test evolution but does not define steps systematically to be followed. We believe that it is essential to understand it first, and the following describes new solution proposals.
- **Co-evolution of production and tests code.** Software is multidimensional, and so is the development process behind it. This multidimensionality lies in the fact that to develop high-quality source code, other artifacts are needed, *e.g.* specifications, constraints, documentation, tests, and so on. Thus, ideally, test code and

production code should be developed and maintained synchronously once newly added functionality should be tested as soon as possible in the development process. Although co-evolution is an important topic, in reusable systems, the test evolution is very incipient. We believe that our study is the step before and will provide the basis for future investigations in the co-evolution area.

1.7 ORGANIZATION OF THE THESIS

This thesis is structured in three parts. Figure 1.2 shows a schematic overview of the thesis structure. Apart from the Introduction Part, the remainder can be outlined in the following way:

Part II - Background. This part provides background concepts on the topics involved in this investigation, as discussed in Section 1.4.

Chapter 2 (Concepts): Basic concepts regarding the topic of this thesis.

Chapter 3 (An overview of the state-of-the-art and related work): Additional concepts regarding topics of this thesis proposal and related work.

Part III - Empirical Studies. This part represents the core of this investigation. It comprises a set of empirical studies to understand test evolution and provides a set of observations and implications that contribute to the advance of the research concerning test evolution in reuse environments.

Chapter 4 (A case study on Test Evolution in an extensive configurable system): We analyzed the test suite evolution of a large configurable system and provided a set of thirteen observations that indicate the effort to develop and maintain the test suite. Additionally, we unveiled how the variability affects the tests.

Chapter 5 (A comparative study in Test Evolution in a set of configurable systems and Single Systems): We evaluated the test evolution in 18 open-source projects from various sizes and domains in configurable systems and their similarities and differences to 18 SS projects. The comparison allows us to identify similarities that can help us adapt techniques from SS to configurable systems and vice versa. The differences can help us design new strategies addressing the discrepancies.

Chapter 6 (An extended study on Test Evolution in a Software Ecosystem): We continued to analyze the test suite evolution in another category of reusable systems (ecosystem) to verify whether some observations are recurring and gather new data that support the findings.

Chapter 7 (Survey with test contributors): We surveyed test contributors to investigate the test evolution from the development point of view and improve the findings in previous stages.

Part IV - Conclusions. Finally, this part concludes the thesis document.

Chapter 8 (Research Synthesis) This part presents the synthesis and evidence extracted from applying the multi-method approach performed in this thesis. Moreover, the strengths and weaknesses of the multi-method approach and lessons learned are discussed.

Chapter 9 (Conclusions) This part concludes the thesis proposal, with a summary and outlook on further investigation.

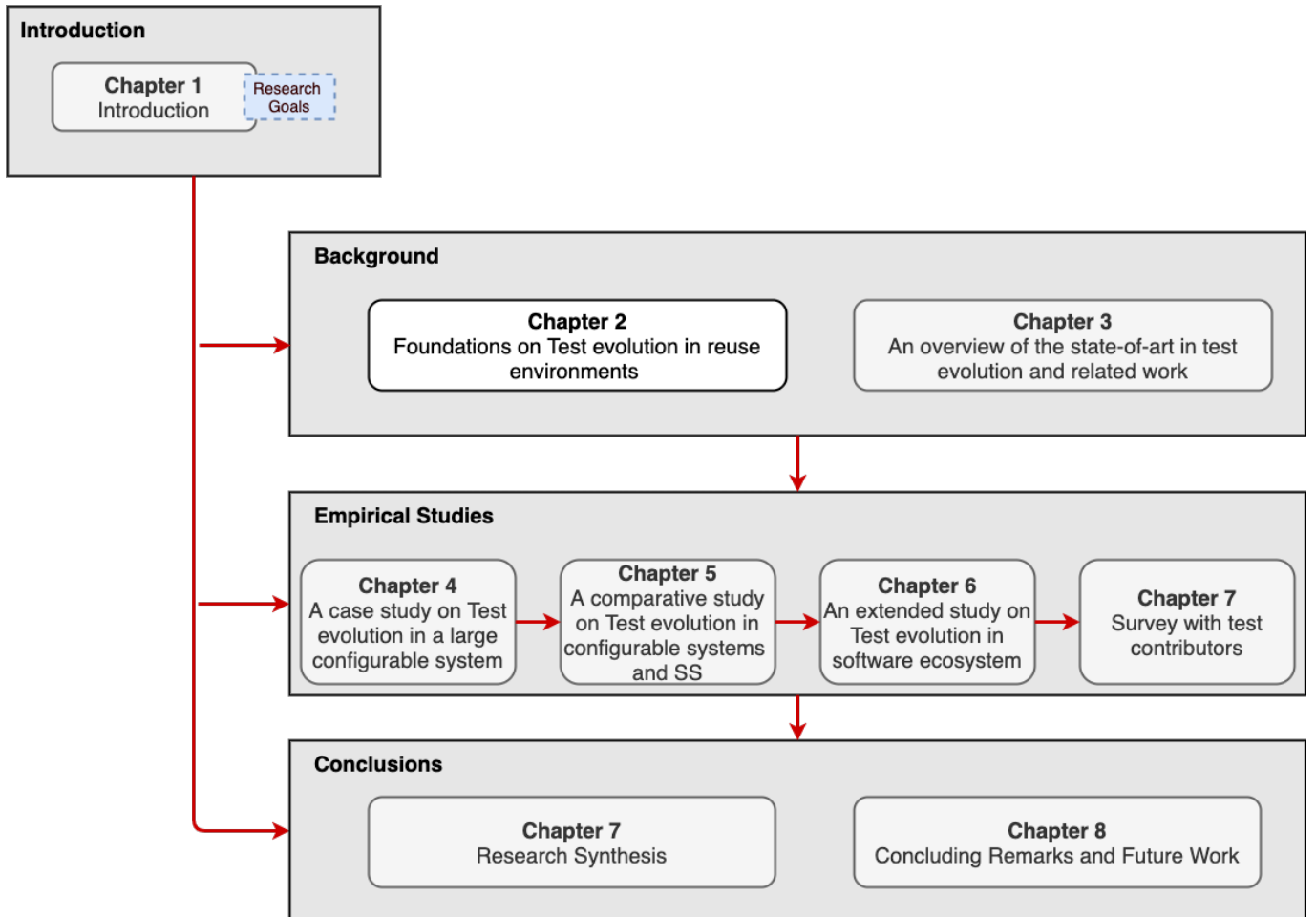


Figure 1.2: Schematic overview of the thesis structure.

PART II

BACKGROUND

MAIN CONCEPTS AND FOUNDATIONS

In reusable systems, software evolution is inevitable since the projects are planned to survive in the long-term [10, 40]. Equally well understood is the necessity of having a good test suite available to ensure the quality of the current state of the software system and ease future changes [5, 41]. However, developing and maintaining a test suite is not straightforward since, in these systems, the variability aspects, such as variation points (elements to manage variability and to facilitate the derivation of different configurations) in configurable systems [10] and dependence management in ecosystems [42], increase the challenges in the development of tests. This way, test evolution has been a challenging problem for decades in reusable environments.

The goal of this chapter is to present the basic concepts in the context of this thesis, and it consists of four main sections: Section 2.1 introduces the *software reuse*. Section 2.2 presents the reusable systems evaluated in this study: *Highly Configurable Systems* and *Ecosystems*. Section 2.3 introduces the *software testing*, and Section 2.4 concludes this chapter.

2.1 SOFTWARE REUSE

The competitiveness of the market and diversification in software development has been a critical issue for employing new engineering practices [43]. Systematic software reuse is one of the most effective software engineering approaches for obtaining benefits related to productivity, quality, and cost reduction [44]. However, many different viewpoints exist about the definitions involving software reuse. Tracz [45] considers reuse as the use of software that was designed for reuse. Basili et al. [46] define software reuse as the use of everything associated with a software project, including knowledge. According to Ezran et al. [47], software reuse is the systematic practice of developing software from a stock of building blocks so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality, and business performance.

In this study, we adopt the general view of software reuse provided by Krueger [48]: “*software reuse is the process of creating software systems from existing software rather than building them from scratch.*” The concept of software reuse and its application present a positive impact on software quality, as well as on cost and productivity [44, 49]. A productivity gain is achieved due to less code that has to be developed, resulting in less testing efforts and saving analysis and design labor, yielding overall savings in cost. The reliability increases since using well-tested components in several systems increase the chance of errors being detected and strengthen confidence in that component. Some reusable systems have proved to be an efficient and effective way to achieve software reuse, such as configurable systems and software ecosystems.

2.2 REUSABLE SYSTEMS

2.2.1 Highly-Configurable Systems (HCS)

A *feature* describes a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option [50]. Products of highly configurable systems (resp. Software Product Lines (SPL)) can be composed by selecting a set of features. Then, HCS can be described as a family of systems created and developed from features. The engineering explores the commonalities and manages variabilities among related products, in which it is possible to establish a common platform on top of software assets that can be systematically reused and assembled into different products.

Based on the selection of features, software engineers can configure distinct products satisfying a range of common and variable features, which comprise both functional and non-functional properties [51]. Additionally, features are usually classified as [52]: (i) mandatory, a feature that must be selected whenever its parent feature is selected; (ii) optional feature, a feature that may or may not be selected; (iii) *OR* feature group, when one or more features in the group must be selected; and (iv) *XOR* (alternative) feature group, when one and only one of the features in the group must be chosen.

2.2.1.1 Variability Implementation in HCS. Variability in source code can be implemented using different approaches, such as: Object Oriented Programming, Aspect Oriented Programming, macro-directives, and so on [10]. Usually in the C language, developers use preprocessor directives. Preprocessor is a language-independent tool for lightweight meta-programming that provides no perceptible form of modularity [53]. Developers use preprocessor directives expressed in terms of conditional compilation macro directives `#ifdefs`¹, whose conditions are essentially `Boolean` expressions over feature names. These directives define whether certain source code fragments should be compiled. For instance, the test code related to `i386` (lines 10 - 20) in the `F00F` test is optional and is included only when configuration option `i386` is enabled.

Listing 2.1: F00F Test Program.

¹For simplicity, we refer to the various conditional inclusion macros, such as `#ifdef`, `#ifndef`, and `#if`, summarily as `#ifdef`.

```

1 #include <signal.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "test.h"
5
6 char *TCID = "f00f";
7 int TST_TOTAL = 1;
8
9 #ifdef __i386__
10 void sigill(int sig){
11     tst_resm(TPASS, "SIGILL received from f00f instruction. Good.");
12     tst_exit();
13 }
14
15 int main(){
16     signal(SIGILL, sigill);
17     tst_resm(TINFO, "Testing f00f instruction.");
18     asm volatile (".byte 0xf0 .byte 0x0f .byte 0xc7");
19     tst_brkm(TFAIL, NULL, "f00f did not properly cause SIGILL");
20 }
21 #else /* __i386__ */
22 int main(){
23     tst_brkm(TCONF, NULL, "f00f bug test only i386");
24 }
25 #endif /* __i386__ */

```

2.2.2 Ecosystems

Software product lines companies increasingly expand their platform outside their organizational boundaries. Once the company decides to make its platform available outside the corporate boundary, the company transitions from a software product line to a software ecosystem [54]. In the last decade, software ecosystems arose as an essential mechanism to promote and support code reuse.

Although companies have different reasons for adopting a software ecosystem approach, one can identify some convincing arguments explaining the current trend: *(i)* increase value of the core offering to existing users, *(ii)* increase attractiveness for new users *(iii)* accelerate innovation through open innovation in the ecosystem, *(iv)* collaborate with partners in the ecosystems to share the cost of innovation, and so on [54]. Some ecosystems are set around packaging platforms for a programming language [42]. Such platforms are built upon the notion of dependencies between packages. Dependence management enables a client package to reuse a particular provider package that implements one desirable feature or a set of features.

2.3 SOFTWARE TESTING

Testing plays an essential role in the quality assurance process for reusable systems, and there are many opportunities for economies of scope and scale in the testing activities [10]. According to Software Engineering Institute (SEI)² in its report on software testing [55], testing is one approach to validating and verifying the artifacts produced in software development. A more detailed definition can be stated as testing is designed to make

²SEI - Software Engineering Institute at Carnegie Mellon University - www.sei.cmu.edu.

sure computer code does what it was intended to do and that it does not do anything unintended.

Testing activities support quality assurance by gathering information about the nature of the software being studied. These activities consist of designing test cases, executing the software with those test cases, and examining the results produced by those executions. As a result, these activities can reduce the risk of failure in the real environment [55].

2.3.1 Testing Levels

In a quality-driven development process, testing activities should be performed along the whole life cycle to find significant problems earlier in the development and thus to avoid resource waste. Early identification of defects is by far the best means of reducing their ultimate cost. The testing activities mentioned are then expressed as testing levels. A different level of testing accompanies each distinct software development activity, and the information for each test level is typically derived from the associated development activity.

The idea behind splitting testing into levels is to build code and test it in pieces and gradually put together into larger and larger portions, to avoid surprises when the entire product is linked together [56]. The general testing levels are following described:

- **Unit Testing:** It is designed to assess the units produced by the implementation phase and is the “lowest” level of testing. Unit testing has a goal of the capability to ensure that each software unit is functioning according to its specification [57]. Also, each unit test must run independently of all other units as well as unit tests must be able to run in any order.
- **Module Testing:** It is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures. As with unit testing, most software development organizations make module testing the responsibility of the programmer [58]. It is possible to describe a process merging module and unit testing objectives in a single level.
- **Integration Testing:** As the units and/or modules are tested, and the low-level bugs are fixed, they are then integrated, and integration testing occurs. It is designed to assess whether the interfaces between units (or modules) in a given subsystem have consistent assumptions and communicate correctly. Integration testing must assume that modules work correctly [58].
- **System Testing:** Its purpose is to compare the system to its original objectives. It assumes that the pieces work individually, and asks if the system works as a whole [59]. This level of testing usually looks for design and specification problems. It is a costly place to find lower-level faults and is usually not done by the programmers, but by a separate testing team [58].
- **Acceptance Testing:** It is designed to determine whether the completed software meets customers’ requirements. Acceptance testing probes whether the software

does what the users want. It must involve users or other individuals who have strong domain knowledge [58].

In this work, we provide a static analysis [23] of the test suite evolution. This way, we only analyzed test artifacts that contain code without the necessity of the system under development. Although we do not make distinctions of the test artifacts based on the testing levels, the analysis comprises test artifacts on the different levels.

2.4 CHAPTER SUMMARY

In this chapter, we presented an overview of the topic investigated in this thesis. We started by introducing the software reuse concepts, its application, and its benefits. We also presented the reusable systems evaluated in this study. Then, we introduced software testing and showed the testing levels.

The next chapter presents additional concepts regarding the topics of this thesis and related work.

AN OVERVIEW OF THE STATE-OF-THE-ART AND RELATED WORK

The goal of this chapter is to describe the basis for understanding the connection between test and evolution fields. Additionally, we discussed the related works on test evolution in reusable systems. The chapter consists of main sections as follows: Section 3.1 describes the connection between test, reusable systems, and evolution; Section 3.2 presents the studies on test evolution; Section 3.2.1 discusses the studies on testing in configurable systems and the open issues in the topic; Section 3.2.2 presents the studies on test in ecosystems; and, finally, Section 3.3 concludes this chapter.

3.1 CONNECTION BETWEEN TEST, REUSABLE SYSTEMS, AND EVOLUTION

Reusable systems aim to support the development of a whole family of software products through systematic reuse of shared assets [60]. Additionally, as they exhibit a long life-span, evolution is an even more significant concern than for single-systems. For many people, evolving a software system has become a synonym for adapting the source code as this concept stands central when thinking of software. Software, however, is multidimensional, and so is the development process behind it [5]. This multidimensionality lies in the fact that to develop high-quality source code, other artifacts are needed. One of these artifacts is the tests, which need to be set up and exercised to ensure quality [5].

Additionally, we have learned that the interplay between software evolution and software testing is often very complex [5, 61]. The interaction that we witnessed works in two directions: software evolution is hindered by the fact that when evolving a system, the tests often need to co-evolve, making the evolution more difficult and time-intensive. On the other hand, many software evolution operations cannot safely take place without adequate tests being present to enable a safety net. This leads to an almost paradoxical situation where tests are essential for evolving software, yet at the same time, they are obstructing that very evolution. Thus, test evolution is an essential topic of research, and it is highlighted as a research idea in the area that is yet few unexplored [5].

3.2 STUDIES ON TEST EVOLUTION

Due to the critical role of tests in ensuring software quality, researchers have investigated test evolution in different contexts [11, 12, 13, 14, 15]. Moonen et al. [5] came across some research ideas in the area of software testing and software evolution that need to be explored. The topics are refinement to the issues that were addressed by Harrold in her “Testing: A Roadmap” [62]. Moonen et al. [5] highlighted that the a posteriori analysis of software evolution, through the mining of e.g., versioning systems, provides a view on how the software has evolved and on how the software might evolve in the future. This kind of analyses was applied in some steps of our work (Chapters 4, 5, and 6).

Studies focusing on co-evolution of production code e related artifacts have been presented in the past [11, 12, 31]. Zaidman et al. [11] investigated two open source projects to understand whether the production code and the tests co-evolve. As a result, they introduced three views: *(i)* the change history; *(ii)* the growth history; and *(iii)* the test quality evolution view. Additionally, they demonstrated the use of these views and distinguished more synchronous co-evolution from a more phased testing approach. The work was later extended [12] with a new industrial case study to contrast with the open-source testing strategies previously investigated. Marinescu et al. [31] conducted an empirical study examining how code and tests co-evolve in six popular open-source systems. It was the first paper that investigated how well patches are covered over a large number of program versions and some of their ideas were used in other studies, such as [63]. Although our study does not analyze the co-evolution, the view of change history and the growth history are quite explored.

Elbaum et al. [64] conducted a controlled experiment and a case study to investigate code coverage and its stability in software evolution. Their results indicated that even small changes during the evolution of a program can have a profound impact on coverage information and that this impact increases very fast as the degree of change increases. The basis of the case study in evolution was applied in one step (Chapter 4) of our work.

Pinto et al. [13] is the first large-scale study of test code evolution. They analyzed 88 program versions, 14,312 tests, and 17,427 test changes from six open source projects in Java based on the proposed technique for studying test-suite evolution and a tool that implements the technique (TestEvol [14]). The results showed that test repair is just one possible reason for test-suite evolution, whereas many changes involve refactorings, additions, and deletions of test cases. The results also indicated that test modifications tend to include complex and hard-to-automate changes to test cases. Mirzaaghaei et al. [65] focused on automating test plans updates. They identified eight scenarios that allow either to repair tests or to use tests to generate new ones and defined test evolution algorithms that automatically repair and generate tests by adapting existing ones. Their results indicated that the approach could successfully repair 90% of the broken test cases and create test cases that cover a large amount of code. The results of these studies motivated us to adopt some metrics to evaluate the effort to develop and evolve the test suite, considering the aspects of the changeability of the code.

To capture the effort necessary to maintain the tests, we need to track test code sub-characteristics. Athanasiou et al. [23] introduces a model that assesses test code

quality by combining source code metrics that reflect three main aspects of test code quality: completeness, effectiveness, and maintainability. In our study, we explored one dimension (maintainability) and provided an evolutionary view of the sub-characteristic of this dimension.

3.2.1 Testing in Configurable Systems

Different strategies for testing configurable systems were summarized in [66, 40, 10]. Runerson et al. [66] performed a systematic mapping study-related to HCS testing. The research indicated that HCS testing is a rather immature area and seems to be a “discussion” topic. Additionally, the study shows that there is a well-established understanding of challenges that includes the analysis of evolution. The mapping shows that 64% of the papers found include proposals, which contain ideas for solutions of the identified challenges, but only 17% of the research report actual use and evaluation of proposals. With a clear picture of needs and challenges, the authors strongly recommend the research community to launch empirical studies, to give a solid foundation for HCS testing in the industry.

Neto et al. [40] presented a systematic mapping study to investigate state-of-the-art testing practices, synthesize available evidence, and identify gaps between required techniques and existing approaches available in the literature. As a result, the study shows that although single-system development approaches have covered several aspects regarding testing, many cannot be directly applied in the HCS context. Also, particular aspects regarding HCS are not covered by the existing approaches, and when the aspects are covered, the literature gives brief overviews. The study reinforces the necessity of investigations that include the analysis of evolution, considering empirical and practical aspects.

Machado et al. [10] performed a literature review of two hundred seventy-six studies. They identified testing strategies that have the potential to achieve these economies, and to provide a synthesis of available research on HCS testing strategies, to be applied towards reaching higher defect detection rates and reduced quality assurance effort. The analysis of the reported strategies comprised two fundamental aspects for HCS testing: the selection of products for testing, and the actual test of products. The findings indicate that the literature offers a large number of techniques to cope with such aspects. However, there is a lack of reports on realistic industrial experiences, which limits the inferences that can be drawn. This study showed a number of leveraged strategies that can support both the selection of products and the actual testing of products.

Medeiros et al. [67] presented a comparative study of 10 state-of-the-art sampling algorithms regarding their fault-detection capability and size of sample sets using a corpus of known configuration-related faults from 24 open-source C systems. In a nutshell, they found that sampling algorithms with larger sample sets are able to detect higher numbers of faults, but simple algorithms with small sample sets, such as most-enabled-disabled, are the most efficient in most contexts. Some ideas related to the construction of the dataset, as well as the implications related to quality, served as the basis in our work. Additionally, the list of open-source configurable systems that are used in our analysis in Chapter 5.

Souto and d’Amorim [68] presented a regression testing technique for configurable systems that explores all dynamically reachable configurations from a test. The results of the evaluation indicated that the method reduced the time by approx. 22% and the number of configurations tested by approx. 45%.

3.2.2 Testing in Ecosystems

Even though tests can mitigate part of the problems discussed in others studies, such as trigger rework in many dependent packages [69], security vulnerabilities [70], and the risk of use of trivial packages [42], the presence and evolution of test suite have not been addressed. Next, we present the researches performed in ecosystems that considered test activities and open issues in the topic.

Axelsson et al. [71] investigated the challenges related to quality assurance in software ecosystems, and identified what approaches had been proposed in the literature. The research method used was a systematic literature mapping, which, however, only resulted in a small set of six papers. They included the “testing” word in the search terms since the testing activity is often associated with quality assurance. The results showed that testing activity is exciting and addresses some valid needs for quality assurance in software ecosystems. As a result, they highlighted the importance of more studies on the topic. Also, they proposed a research agenda in the testing activities that deserve more attention by the community, such as test case selection, automatic test case generation, on-line testing, test infrastructure, test evolution, and so on.

Greiler and van Deursen [72] discuss how to perform testing in plug-in based systems like Eclipse effectively. They present data from interviews with 25 practitioners (from both open source and closed source ecosystems) on their testing practices. The authors then focus on the need for improved test suite understanding, both for individual test cases and for the entire suites, as a consequence of the extensive use of automatic testing. In particular, they address the understanding problem beyond the unit level, to capture interactions between plug-ins and provide an understanding of which plug-ins are exercised in test suites. For software ecosystems, this is a significant challenge since different organizations will develop the various plug-ins and platforms, and there may be limited access to the information needed to achieve understanding. To improve test suite understanding, they propose five architectural views that can be presented to developers. They describe and evaluate an implementation of those views in an IDE.

Abdalkareem et al. [42] performed an empirical study involving more than 230,000 `npm` packages and 38,000 `JavaScript` applications to understand why developers use trivial packages (i.e., packages with less than 35 lines of code and cyclomatic complexity of 10). The authors show that client package developers perceive trivial packages as well-implemented and tested providers. However, the authors identified that, contrary to client developers’ perception, only 45.2% of the trivial packages have any tests. Also, the authors observed that testing in trivial packages is not different from testing in non-trivial packages.

Kula et al. [73] investigated the relation between clients packages’ trust in provider packages and the update rate of such providers. Trust involves the assumption of both

Table 3.1: Comparison of the characteristics evaluated in our study and the related works.

Related Works	Characteristics							
	Tests	Literature Review	Evolution	Effort	Maintainability	Variability	Ownership	Mining Systems
Moonen et al. [5]	X		X					
Zaidman et al. [11]	X		X	X	X			X
Marinescu et al. [31]	X		X	X	X			X
Elbaum et al. [64]			X					X
Pinto et al. [13]			X		X			X
Mirzaaghaei et al. [65]				X				
Athanasiou et al.	X		X	X	X			
Neto et al.	X	X				X		
Machado et al.	X	X				X		
Medeiros et al.						X		X
Souto and d'Amorim	X					X		
Axelsson et al.	X	X						
Abdalkareem et al	X						X	X
Kula et al.			X					X
Claes et al	X							X

functional and non-functional correctness, *e.g.*, client package maintainers need to trust the reliability of non-functional attributes such as security and stability of an adopted library. The authors identified that a client package maintainer might not believe the test maturity of the latest provider version. Also, the authors highlighted that the client package maintainers are suspicious about the rigor of internal testing. Client package maintainers believe that some bugs cannot be uncovered during in-house testing, but only as post-release defects. In this case, client package maintainers search for the most stable version (in many instances deemed as the popular) release. These results motivated us to investigate the presence of tests in popular packages and the evolution in them. This analysis can confirm if the concern about the realization of the tests and quality of providers packages hold.

Claes et al. [74] studied the errors in the context of the CRAN archive, a long-lived software ecosystem consisting of over 5000 R packages being actively maintained by over 2500 maintainers. Based on an analysis of package dependencies and package status, they presented preliminary results on the sources of errors and the time that is needed to fix them. Despite this study checked the tests, they are not considered in the analysis.

3.2.3 Studies Comparison

Table 3.1 shows a comparison of the characteristics evaluated in our study and the related works. This table shows how our work is different from other studies. In this context, as far as we know, our research is the first attempt to analyze the test evolution in reusable systems with special attention to the effort, maintainability, and ownership. Moreover, our research methods combine findings from different studies, making them more reliable and generalizable. The synthesis of empirical evidence in the test evolution process will eventually lead to better tool support and evolution principles.

3.3 CHAPTER SUMMARY

This chapter presented the basis for understanding the connection between test, reusable systems, and evolution, discussing these topics' interplay. Moreover, it provides an overview

of the existing research related to the subjects of this thesis. Finally, we also presented related work similar to our investigation.

In the next chapters, we present a set of studies on test evolution in reusable systems and provide empirical evidence by analyzing open-source projects from various domains and sizes.

PART III

EMPIRICAL STUDIES

TEST SUITE EVOLUTION IN A HIGHLY-CONFIGURABLE SYSTEM: A CASE STUDY ON THE LINUX TEST PROJECT (LTP)

In the previous chapter, we provided an overview of the existing research related to this thesis's subjects and presented related work to our investigation. The goal was to map out the test evolution field in reusable systems, synthesize available evidence to suggest important implications for practice, and identify research trends, open issues, and areas for improvements. We identified that the existing research in reusable systems had focused much of its efforts on variability evolution as it occurs in the variability model, but it has ignored the evolution of other related artifacts [75, 76, 77, 78, 79, 80]. The existing studies covering variability evolution across different artifacts focus mainly on production code and build systems [19, 12]. The lack of a thorough understanding of how tests evolve is directly reflected in the absence of existing tools to support the test evolution process and methodologies that help the development teams in the maintenance of the test suite [5, 21, 10].

Thus, additional systematic reports are necessary to provide more evidences about the test evolution in reusable systems. In this context, this chapter presents a case study to analyze the test evolution of a large configurable system. The objective of this chapter is to gain a better understanding of the test evolution, evaluating characteristics that indicate the effort to develop and maintain the test suite, and unveiling how the variability affects the tests. To achieve this objective, we mined the versioned history of the Linux Test Project (LTP)¹, the functional and regression test suite for testing the Linux Kernel. Over the years, LTP has become a focal point for Linux testing and Linux testing tools [81]. The project encompasses a large number of test cases which cover multiple platforms for validating the reliability, robustness, and stability of the Linux kernel [82].

In particular, we provide a static analysis [23] of the test suite, with special attention to the four main directories of the project that reflect the subsystems of the Linux

¹<https://linux-test-project.github.io>

Kernel. To keep the uniformity and facilitate the correlation with Linux Kernel, we refer the directories of the project as subsystems. Software practitioners and researchers increasingly recognize the importance of investigating the modularity and the changes that occur over, to get insights regarding evolution in the different parts of a project [83, 25]. We used the GQM (Goal Question Metric) approach [84] to establish our goal, questions, and metrics, assessing 16 years of development, 122 releases, more than 400,000 test cases and 63 millions of lines of code.

The remainder of the chapter is organized as follows: Section 4.1 provides an overview of LTP; Section 4.1.2 presents the methodology defined to gather data from LTP, emphasizing the data collection process; Section 4.3 describes the results of the study and shows a set of 13 observations about the effort of development, maintainability, changes, and variability in a test suite during its evolution; Section 4.4 presents the discussion of the results and the summary of findings; Section 4.5 presents threats to validity, and finally Section 4.6 presents a summary of the chapter;

4.1 BACKGROUND

4.1.1 The Linux Test Project (LTP)

The Linux Test Project (LTP) is a well established open source project that aims to bring test automation to a large highly-configurable system: the Linux kernel [81, 82]. The LTP project delivers test suites to the open source community to validate the reliability, robustness, and stability of the Linux kernel. Although functional and stress tests have their importance in the LTP, this project focus on regression tests [85]. Regression tests are responsible for testing the conformance after modifications [86]. The dominant programming language in LTP is ANSI-C, but there is code written in Perl, and shell scripting languages. Moreover, LTP is relatively large, and its content has historically varied in quality and code coverage due to its size [87].

4.1.2 LTP Structure

The LTP test suite is designed to be easy to use, portable, and flexible [82]. To achieve this goal, the LTP structure follows a well-defined organization, as can be seen in Figure 4.1. The test cases are organized in *test suites*, *i.e.*, a text file containing one test case per line. *Test suites* are usually stored under the *runtest* directory of LTP and they are a convenient way of grouping test programs together to create custom test suites. Test programs correspond to the source code of test cases. To make it easy to find test programs (.c files), they have been organized under a folder structure represented by *testcases subsystem* that partially reflects its architecture. In each release, the *testcases subsystem* vary in the number of folders at the first level. In this study, we focused on four folders: **Kernel**, **Network**, **Misc**, and **Commands**. These folders are available since the initial releases, and they implement the main test programs.

- **Kernel** - Inside this subsystem, each kernel part is in its own folder. This subsystem contains tests for the Kernel, such as: filesystems, io, ipc, and system calls;

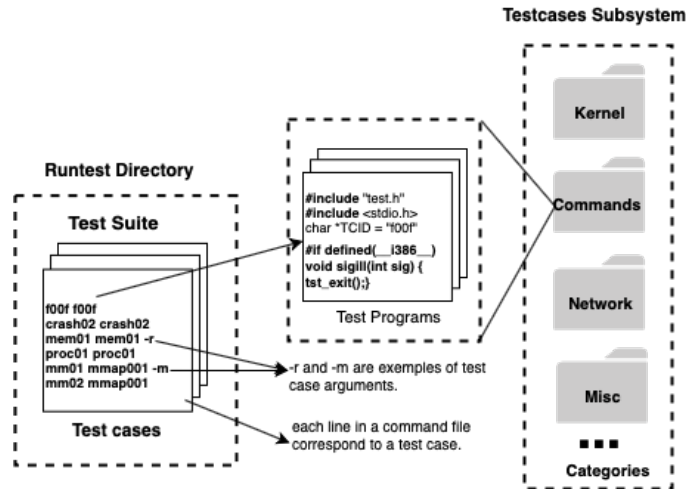


Figure 4.1: LTP Organization.

- **Commands** - This subsystem contains tests for user-level commands. These tests evaluate the commands commonly used in application development, such as: `ar`, `ld`, `ldd`, `nm`, `objdump`, and `size`;
- **Network** - The network subsystem contains tests for `ipv6`, `multicast`, `nfs`, `rpc`, `scpt`, and network related user commands; and
- **Misc** - The misc subsystem contains the tests that do not fit into one of the other categories.

4.1.3 Basic test structure

A test case is a single action that generates a result usually restricted to **PASS/FAIL**. Listing 4.1 shows a test program that executes a simple test case for handling the **Pentium F00F** bug. This bug is a security flaw of this processor's family that allows an unprivileged user to lock the processor. It is an example of a catastrophic test case. If the system does not correctly handle this test, it will likely lock up.

LTP was designed to be flexible enough to allow test programs to be added to it without requiring the use of any cumbersome features that are specific to a certain test driver [82]. Each test program includes the `test.h` header (line 4), that provides a small set of functions used to help with the consistency of test programs and to act as a convenience for the developer. Some of the functions in LTP make use of global variables that define various aspects of the test case. The test program name is defined in the `TCID` variable (line 6). This name is used as an identification for the tests results and, in many cases, it is the same as the test case. The global variable `TST_TOTAL` (line 7) is of type `int` and should be used to specify the number of individual test cases within the test program.

Listing 4.1: F00F Test Program.

```

1 #include <signal.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "test.h"
5
6 char *TCID = "f00f";
7 int TST_TOTAL = 1;
8
9 #ifdef __i386__
10 void sigill(int sig){
11     tst_resm(TPASS, "SIGILL received from f00f instruction. Good.");
12     tst_exit();
13 }
14
15 int main(){
16     signal(SIGILL, sigill);
17     tst_resm(TINFO, "Testing f00f instruction.");
18     asm volatile (".byte 0xf0.byte 0x0f.byte 0xc7");
19     tst_brkm(TFAIL, NULL, "f00f did not properly cause SIGILL");
20 }
21 #else /* __i386__ */
22 int main(){
23     tst_brkm(TCONF, NULL, "f00f bug test only i386");
24 }
25 #endif /* __i386__ */

```

Variability in the test code is expressed in terms of conditional compilation macro directives `#ifdefs`² (line 9), whose conditions are essentially `Boolean` expressions over feature names. These directives define whether certain source code fragments should be compiled. For instance, the test code related to `i386` (lines 10 - 20) in the F00F test is optional and is included only when configuration option `i386` is enabled.

4.2 METHODOLOGY

This section presents the research design and the data collection process to assess the LTP data. We provide a static analysis of the tests during evolution. According to Athanasiou et al. [23], a static analysis is preferable in open source projects for two main advantages. First, this type of analysis does not require to compile the source code of open source systems. It is an important advantage since compiling the source code of open source systems can be very hard due to missing libraries or because a special version of a compiler is necessary [12]. Second, the static analysis does not require the execution of the test suite, a task that is time-consuming [23, 11].

4.2.1 GQM Model

In this study we employed ideas from goal-oriented measurement to help us to find metrics for analyzing the test evolution in an HCS. We followed the GQM (Goal Question Metric) approach to state the research goals and derive corresponding metrics from them [84].

²For simplicity, we refer to the various conditional inclusion macros, such as `#ifdef`, `#ifndef`, and `#if`, summarily as `#ifdef`.

Figure 4.2 shows the GQM model employed in this study. The main goal is to analyze the test suite evolution in a large highly-configurable system (G1). To archive this goal we focus on two main points: (i) Development effort, and (ii) and Maintainability (G3) considering general ((G4) [23] and variability aspects (G5) [88, 89].

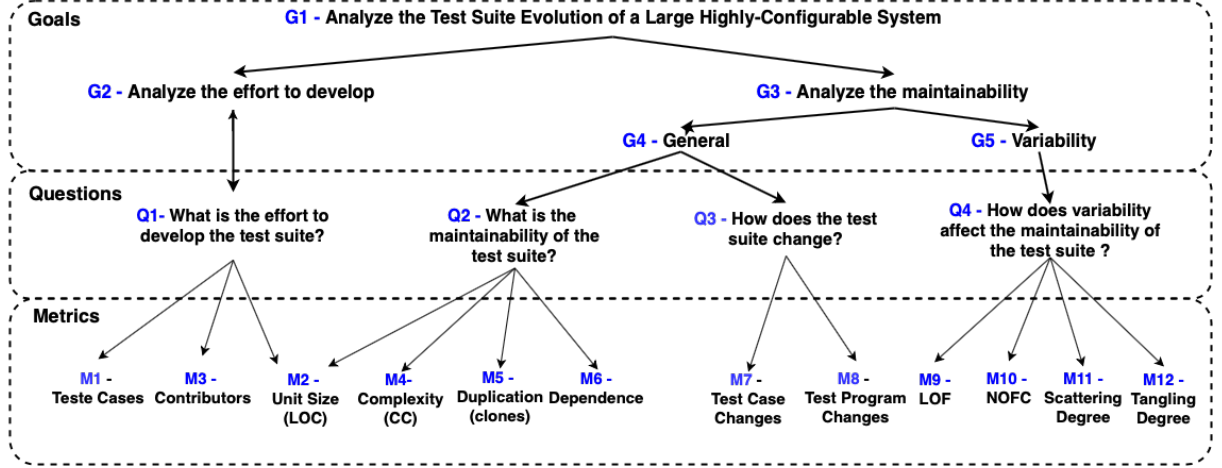


Figure 4.2: GQM model of the study.

4.2.2 Research Questions

To achieve the goal and derived sub-goals, we defined the following questions:

RQ1 - What is the effort to develop the test suite?

To answer **RQ1**, we used the measurements of maintainers' effort suggested by Zhou et al. [25]. This work observed that as more files a developer oversees, the more time and effort he will need to devote. Moreover, the number of contributors implies effort and productivity. Thus, we use the number of test cases maintained, the unit size (LOC), and the number of authors in the commits during a period (month) to characterize the amount of work and consequently the effort.

RQ2 - What is the maintainability of the test suite?

To answer **RQ2**, we adopted the maintainability sub-characteristics of the test code quality model: unit size, unit complexity, unit dependence, and duplication [23]. The test code quality model is based on the Software Improvement Group (SIG) quality model, operational implementation of the maintainability characteristic of the software quality model defined in the ISO/IEC 9126 [90].

RQ3 - How does the test suite change?

To answer **RQ3**, we analyzed the changes made in the LTP test cases. According to Athanasiou et al. [23], changeability and stability are clearly aspects of test code maintainability and provide valuable information about maintenance [91, 92].

RQ4 - How does variability affect the maintainability of the test suite?

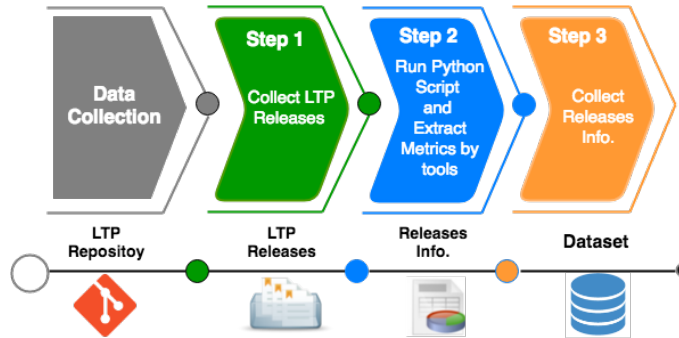


Figure 4.3: Data collection process.

To answer **RQ4**, four well-known variability metrics were collected: Lines of Feature Code (LOF), Number of Features Constants (NOFC), Scattering Degree (SD), and Tangling Degree (TD) [93, 94, 88]. We analyzed if the test artifacts that contain variability are growing, becoming overly complex, and impacting in the project maintainability during the evolutionary history.

4.2.3 Data Collection

Figure 4.3 shows the data-collection process. The process comprises three steps: for step 1, LTP releases were collected in the repository. All the LTP releases (up to version `ltp-201504200`) are publicly available at Sourceforge. Also, there is a Git repository on GitHub that is frequently updated. Depending on how far it is from the previous release, it contains a few tens of new test cases and hundreds of fixes. In step 2, we ran a set of script on each release. A Python script generates information about the release, such as: subsystems, number of test cases, the name of the test case, and test code measurements. Finally, in step 3, the data collected was stored in a relational database. The database provides a concrete testbed for future analyses. All steps in our process are fully automated and supported by tools.

4.3 RESULTS

This section presents the results of the study after applying the methodology on the available raw data. A total of 122 releases were analyzed involving sixteen years of development.

4.3.1 RQ1 - What is the effort to develop the test suite?

A set of metrics was collected to analyze the relative and absolute growth of LTP, providing data history and the effort necessary to develop the project during evolution. Each metric and the main findings are discussed next:

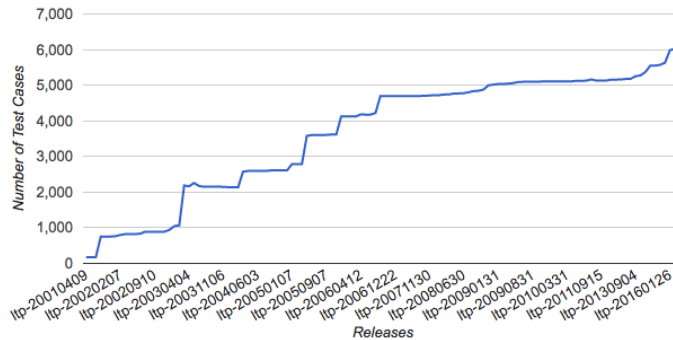


Figure 4.4: Number of Test Cases per Release.

4.3.1.1 M1 - Test Cases: We analyzed the test case history to verify how the LTP evolved and evaluated the effort to develop the test suite. The number of test cases is an essential metric in this direction since more test files a developer oversees, more time and effort he will need to devote [25]. Moreover, each file under maintenance may need to be considered when fixing a bug even if it is ultimately not changed.

Figure 4.4 shows the amount of test cases over the 122 releases. It is important to highlight that the 122 releases are represented in the figure, but the name of releases shown on the X-axis is a subset of the total. We can observe that the number of test cases has grown in the last sixteen years. The first release (`ltp-20010409`) had a total of 176 test cases, while the last release analyzed (`ltp-20160920`) is composed of about 6000 test cases. Even with the reduction in the number of test cases between the last two releases, we can observe a tendency of growth over time.

Initially, LTP experienced fast growth in the number of test cases. Considering the relative increase, it indicates that more effort is spent in the early stages of the project. Figure 4.5 shows the evolution of test programs per subsystem. We used a logarithmic (log) scale in the vertical axis. This way, we emphasized the curve growth, since the values for some subsystems are discrepant during the evolution. We can observe that the test programs are mainly concentrated in the `Kernel` subsystem. Likewise the `Kernel`, the `Network` presents a consistent growth. Interestingly, after a sudden growth, in the beginning, the `Misc` subsystem presents stable behavior. The `Commands` has the same number of test programs from the first to last release, with slight variations during evolution.

Analyzing the LTP evolution for test suites, we can highlight some information. Figure 4.6 shows that the number of test suites has increased over the years. However, the average number of test cases per test suite presented a mix of sub-linear and super-linear growth. Interestingly, the effort to develop the project is mainly concentrated in a few test suites. For example, in version `ltp-20160510`, the six major test suites have 61% of all test cases: `syscalls` (18%), `ltp-lite` (13%), `stress.part3` (11%), `network_stress.whole` (7%), `network_stress.tcp` (5%), and `controllers` (5%). This can be explained, since the `syscalls` test suite is related to the `Kernel` subsystem and the `network` test suites are related to `Network` subsystem, that are large and important modules of the Linux Kernel resulting in more test cases for these parts.

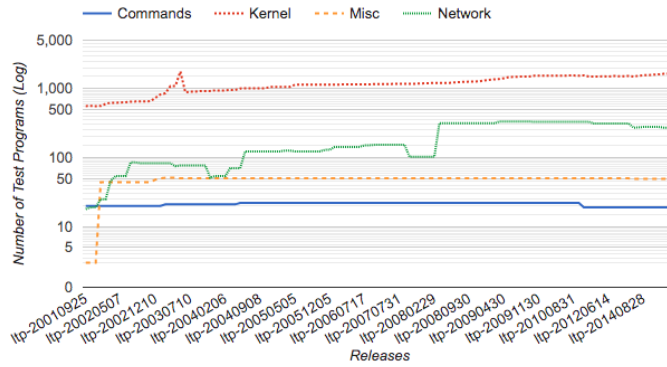


Figure 4.5: Test Programs evolution per subsystem.

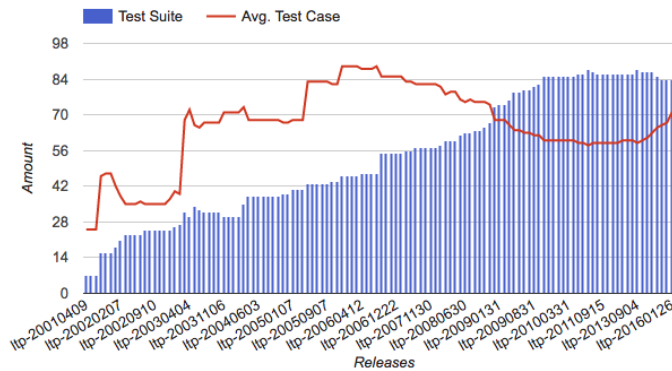


Figure 4.6: Quantity of Test Suites and avg. of test cases in each one.

Observation 1. *LTP presents continuous growth of the number of test cases, resulting in more effort to develop the test suites during evolution. However, the effort to develop the project is mainly concentrated in a few test suites, such as: `syscalls`, `stress`, and `network`. Also, we observed that the test programs are mainly concentrated in the Kernel subsystem.*

4.3.1.2 M2 - Unit Size: Unit size is measured as the number of lines of code (LOC) in a unit (test program) [23]. LOC is typically used to predict the amount of effort that will be required to develop a project [95] as well as to estimate programming productivity or maintainability in test code once the code is produced [23].

Figure 4.7 shows the number of LOC of each LTP release over the years. Our study analyzed more than 63 millions of lines of code. We can observe that LOC increased significantly from the initial to the current release. This growth indicates a constant effort of development. Interestingly, the growth curves sometimes exhibit discrete jumps, *e.g.* from `ltp-20010801` to `ltp-20010925`. It can be traced to the inclusion of 11 new test suites, such as the `syscalls` that added 523 new test cases in version `ltp-20010925`, and the `tcp_cmds` test suite that added 16 new test cases. Such additions may be taken

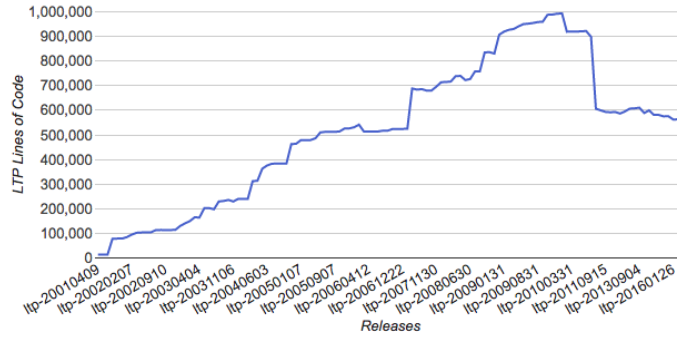


Figure 4.7: LOC per releases.

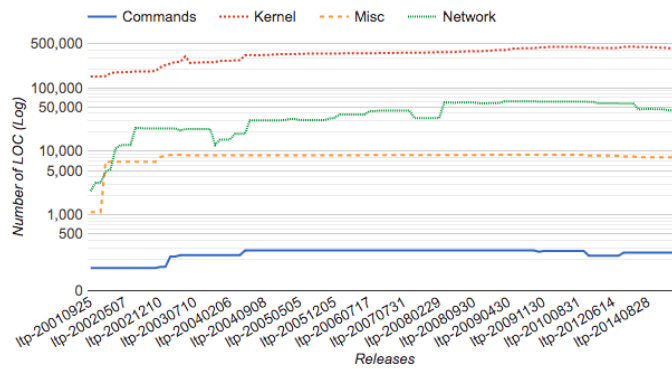


Figure 4.8: LOC in each subsystem per releases.

as an example of punctuated evolution [96], as opposed to progressive evolution.

Comparing the growth behavior of LOC with other studies, Capiluppi et al. [97] analyzed several open-source projects and identified that code growth is generally linear. Smith et al. [98] reported that some projects also exhibited periods of stability or even declining size, but still growing in general. Israeli et al. [99] analyzed the Linux kernel growth using LOC, and found a mix of sub-linear and super-linear growth. In spite of we had examined test code, a different set of code from all the ones cited here, our results also support a mix of code growth.

Additionally, we explored the LOC growth in four subsystems of the LTP architecture. Figure 4.8 shows the number of LOC over time for different subsystems. Similarly to the overall LTP project, the subsystem presented a mix of super-linear and sub-linear growth. While the *Network* and *Misc* subsystems appear to have a decreasing trend in the last few years, the *Kernel* subsystem presented consistent growth for the entire period.

Observation 2. *LTP presented a mix of sub-linear and super-linear code growth. In general, the size of LOC has increased significantly from the initial to the current release, resulting in more effort to develop and evolve the project. Moreover, the *Kernel* subsystem presented consistent growth and concentrate the major effort of project de-*

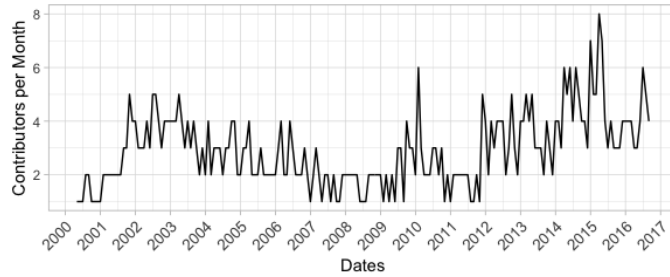


Figure 4.9: Number of Contributors per month.

velopment for the whole period.

4.3.1.3 M3 - Contributors: We analyzed the number of contributors involved in test activities in each month during the project evolution, as can be seen in Figure 4.9. The X-axis presents the years without the months, once the figure does not have sufficient space for all months in each year. We observed that the average of contributors in each month is 2.9, but in some months *e.g.*, 2007-01-01, there was just one contributor. Also, we analyzed the contributors per subsystems. In general, the number of contributors remained stable in all analyzed subsystems except in the `Kernel`. As we expected this subsystem presents the highest mean 2.6 of contributors per month (more about these data can be seen in Table 4.1) since it is the biggest subsystem considering number of the test case and LOC.

Additionally, we analyzed the workload of the contributors involved with test activities over time. We used Man-Month (M/M) to measure workload [100]. Thus, we extracted the number of test lines of code (LOC) modified in a month and divide by unique contributors to measure Man-Month. We observed high jumps that happened in the years 2001, 2004, 2005, 2007, 2011, and 2013. Looking at these years more closely (Figure 4.10), we can observe that some months present outliers. Investigating these particular months, we noted that the outliers are related to integration in the master branch of the project and repository migration, making the values surprisingly high. Therefore, we decided to exclude the outliers by applying the Univariate approach. The outliers are those observations that lie outside $1.5 * IQR$, where IQR , the ‘Inter Quartile Range’ is the difference between 75th and 25th quartiles. Figure 4.11 shows the Man-Month evolution without the outliers. A logarithmic (log) scale was used in the vertical axis to emphasize the curve growth, since the average values of Man-Month present significant variations during the evolution. In spite of we get more realistic values after to exclude the outliers, in general, the mean of Man-Month in LTP is 17039 LOC per contributor, and it can be considered high [100]. High values of Man-Month throws an alert since a high workload can contribute to a low quality of code.

Observation 3. *The number of contributors is very irregular during evolution. Concerning subsystems, only the `Kernel` presents the growth in the number of contributors,*

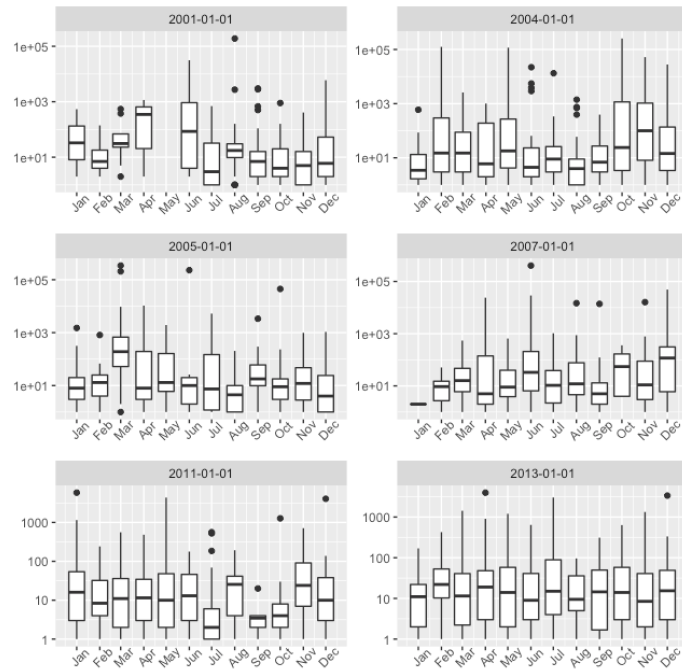


Figure 4.10: LOC per man-month (log10).

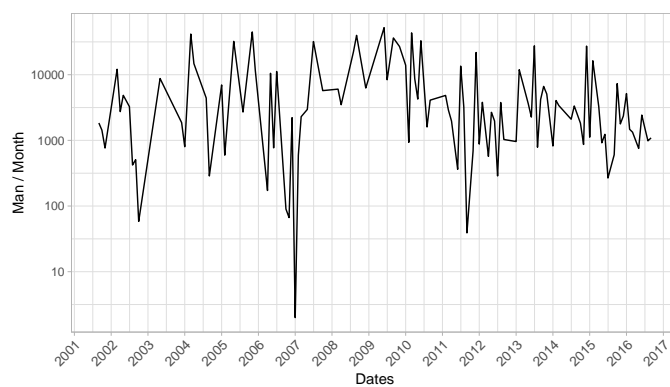


Figure 4.11: The productivity per Man-Month.

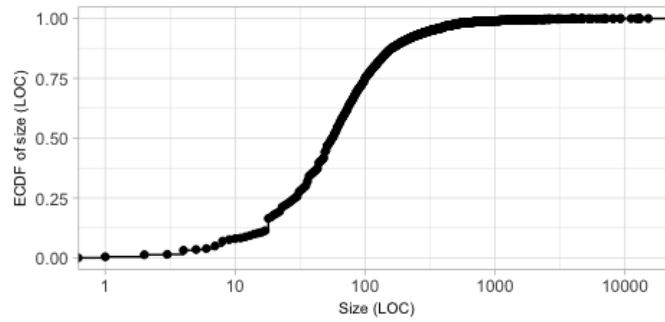


Figure 4.12: Percentiles of Test Program (LOC).

and all other subsystems exhibit stable values. Additionally, the LTP contributors presented a high workload during evolution, that deserves special attention by the development team since this situation can result in a low quality of code.

4.3.2 RQ2 - What is the maintainability of the test suite?

As a measurement of the maintainability we used existing maintainability measuring metrics proposed by Athanasiou et al [23]. These metrics are based on Software Improvement Group (SIG) quality model which is an operational implementation of the maintainability characteristic of the software quality model defined in the ISO/IEC 9126 [23].

4.3.2.1 M2 - Unit Size: The relation between unit size (LOC) and maintainability is recognized both in the context of production code and test code [23]. According to Athanasiou et al. [23], as the LOC increases, it becomes harder to analyze. Figure 4.7 shows the growth curves of unit size in LTP. We can observe some impressive jumps in the growth curves. Analyzing these jumps more closely, we can observe that they are related to maintenance activities. For example, the most notable jump was observed from release `ltp-20101031` to `ltp-20110228`, and it is related to activities of source code clean-up/maintainability. These activities perform modifications or exclusions to improve usability and maintenance. For example, the exclusion of 1.199 code files in release `ltp-20110228` resulted in a significant decrease of 152K lines of code.

Figure 4.12 shows the distributions of unit size using an empirical distribution function (ECDF). The value in this distribution at any specified point of the measured variable (LOC of test programs) is the fraction of observations of the measured variable that are less than or equal to the specified value. This way, using the thresholds suggested by Athanasiou et al. [23], we can observe that 21.6% of LTP test programs present a unit with a size of less than 24 (lines of code), *i.e.* easy to maintain. Moreover, 5.4% of the tests programs are classified having a moderate values ($24 < \text{unitsize} \leq 31$) and 14.5% classified as high values (unit size between 31 and 48). However, 58% of the test programs are classified as being very high values (unit size greater than 48), indicating that the maintenance of tests can become difficult.

The high percentage of test programs with high values of unit size (LOC) could be a warning for the Obscure Test, and the Eager Test code smells [101] in the project. An obscure test is a test that has a lot of noise in it, noise that's making it hard to understand, and the consequences are that such test is harder to maintain and it does not serve as documentation. An eager test attempts to test too much functionality.

Observation 4. *The unit size growth curves seem to be slowing down since 2011, due to activities of source code clean-up/maintainability. However, a high percentage of the test programs are classified as being very high values of unit size indicating that the maintenance of tests can become difficult.*

4.3.2.2 M4 - Unit Complexity: The complexity is something that should be kept as low as possible to avoid writing tests for test code [23]. This is also underlined in the description of the Conditional Test Logic code smell [101], which advocates keeping the number of possible paths as low as possible to keep tests simple and correct. High unit complexity is, therefore, affecting both the analysability and the changeability of the test code. To measure LTP unit complexity, we used McCabe's Cyclomatic Complexity ($v(G)$) [102, 103], as proposed in the test quality model in [23]. Complexity is a single quantity that estimates the control flow of the code [102]. For a single function, the unit complexity is equivalent to the number of conditional branches in the program [102]. In the C language, the control flows include `if-then-else`, `for`, `while`, `do`, and `case` statements.

The results for complexity applied to the full code base of all LTP releases can be seen in Figure 4.13. The complexity values presented a mix of sub-linear and super-linear growth. As may be expected, when the size of the code grows, so does the total complexity [99, 104]. Moreover, we analyzed the unit complexity metric for normalized values, such as the average of the unit complexity per function, as shown in Figure 4.14. In general, using the threshold for test code originally suggested by Athanasiou et al. [23], in the average, the LTP functions test cases present a very high values (*unit complexity* > 4) of complexity. However, since 2007, the values of unit complexity indicates a declining trend. Thus, a total of unit complexity is, in general, growing slower than the number of functions, and the average complexity is decreasing.

Moreover, we explored the Cyclomatic Complexity results by subsystems, as shown in Figure 4.15. While `Network`, `Misc`, and `Kernel` subsystems presented a mix of super-linear and sub-linear growth, the `Commands` subsystem, which contains test cases in order to check if required command of Linux Kernel exist, has slightly increased in the entire period. The significant growth in the complexity values from release `ltp-20040405` to `ltp-20040506` can be attributed to `Kernel`. Moreover, the improvements in the last years are linked with the `Kernel` and `Network` subsystems. Other interesting aspect is that the `Commands` subsystem presents better unit complexity values than `Network`, `Misc`, and `Kernel` subsystems. It can be related to the subsystem presents fewer test cases and LOC that the other ones, facilitating the quality aspects of test code. In addition, we can highlight that the `Network` subsystem presented the highest average values during evolution. In the same way that other studies [99, 104], the unit complexity behavior for

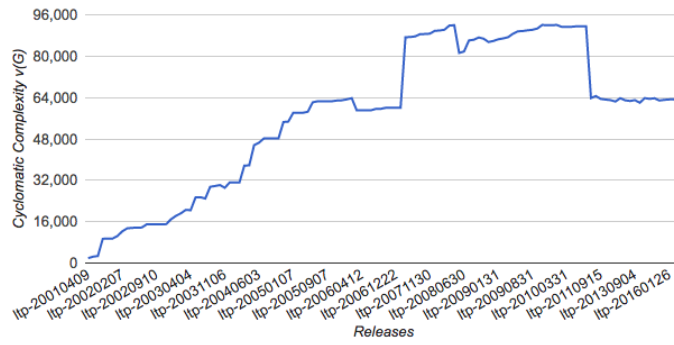


Figure 4.13: Accumulated Cyclomatic Complexity per Releases.

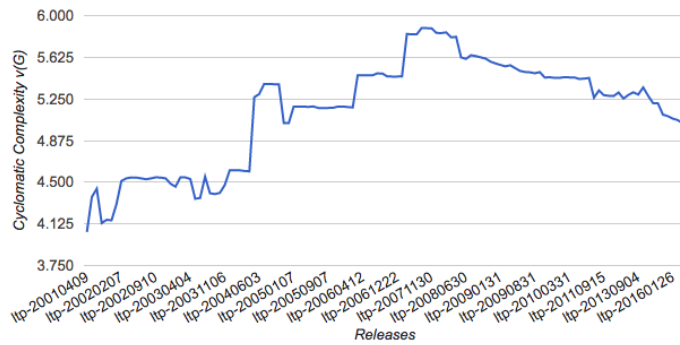


Figure 4.14: Average of Cyclomatic Complexity per Function.

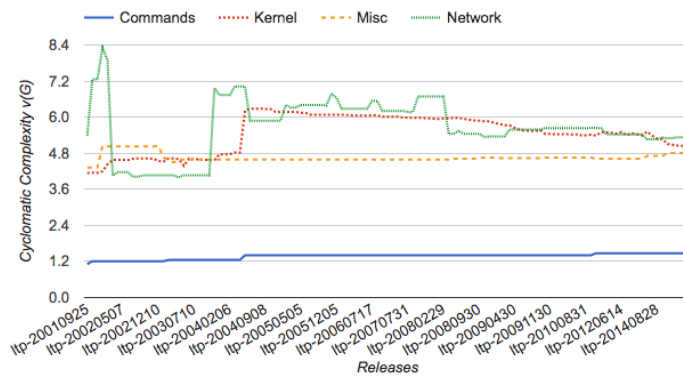


Figure 4.15: Average of Cyclomatic Complexity per Subsystem.

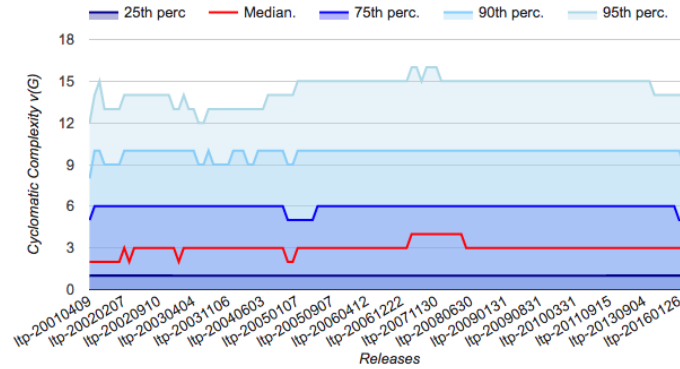


Figure 4.16: Percentiles of unit complexity per Function.

each subsystem follows those shown in LOC (Figure 4.15).

Figure 4.16 shows the distributions of unit complexity and their evolution of percentiles. Using the thresholds suggested by Athanasiou et al. [23], we can observe that 29% of the LTP functions presented low values ($unit\ complexity < 1$) during the evolution, and 15% of the test functions presented moderate values ($1 < unit\ complexity \leq 2$). However, the most part of functions present high values (21%, $2 < unit\ complexity \leq 4$) or very high values (35%, $unit\ complexity > 4$). Also, we found there is no improvement of the functions with high unit complexity from the first to the last releases. Initially in the `ltp-20010409` release, 46% of the functions had high values, but in `ltp-20160920` it is about 54%.

Observation 5. *The Unit Complexity metric has indicated that the majority of functions (56%) in LTP present a very high values ($unit\ complexity > 4$), i.e., they are not easy to maintain. Regarding subsystems, the **Commands** subsystem has slightly increased the unit complexity average in the entire period. In addition, the improvement of the unit complexity values in the last years are linked with **Kernel** and **Network** subsystems.*

4.3.2.3 M5 - Duplication: is measured as the percentage of all code that occurs more than once in at least 70% identical code blocks with more than six lines (ignoring white lines) [105]. Test code duplication occurs when copy-paste is used as a way to reuse test logic. This results in many copies of the same code, a fact that may significantly increase the test maintenance cost and make the evolution significantly more difficult [23, 105]. Additionally, test code duplication is a code smell [23], and affects changeability since it increases the effort that is required when changes need to be applied to all code clones. It also affects stability, since the existence of unmanaged code clones can lead to partially applying a change to the clones, thus introducing logical errors in the test code.

In this study, we focus on function clones. This way, we extracted the total, and the percentage of cloned functions in each release, as can be seen in Figure 4.17. The LTP presents high percents (at least 40%) of cloned functions per release. Some releases present values higher than 50%, and we can say that these releases have more cloned functions than non-cloned functions. Such releases have a high update anomaly risk; every update

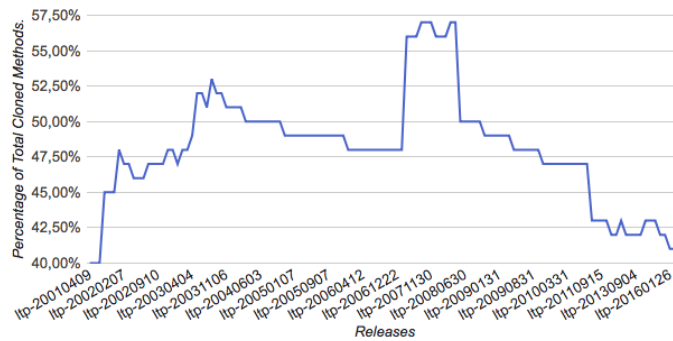


Figure 4.17: Percentage of Total Cloned Functions.

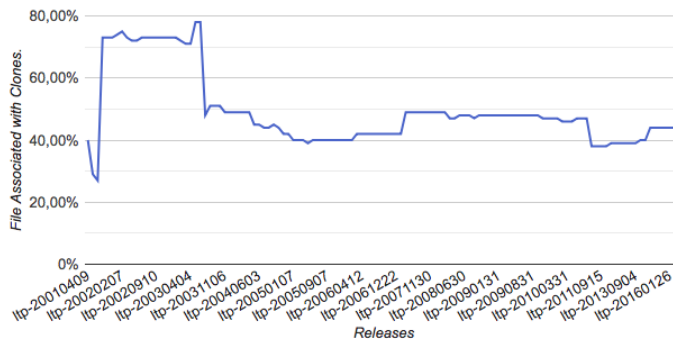


Figure 4.18: Percentage of File Associated with Clones.

to the project has a higher chance of involving a clone than not. However, in the last years, LTP presented a decreasing trend of cloned functions, this can be attributed as a result of clean-up/maintainability on the source code, that was described in M2 - Unit Size.

While the number of cloned functions give the overall cloning statistics for a subject release, it cannot provide any clue as to whether the clones are from some specific files or scattered all over the release among many files. This way, we also analyzed the number of files associated with clones in each release [105]. We consider that a file is related to clones if it has at least one function that forms a clone pair with another method in the same file or a different file. Figure 4.18 shows the percentage of files associated with clones of a release. In general, LTP presents the lowest values of files related to clones in the first three releases (between 27% and 40%) reaching almost 80% in the next releases. However, since 2003 the values are stable at around 40%. From a software maintenance point of view, a lower value of files associated with clone percentage is desirable, as in this case, clones are concentrated in certain specific files and thus may be easier to maintain [105].

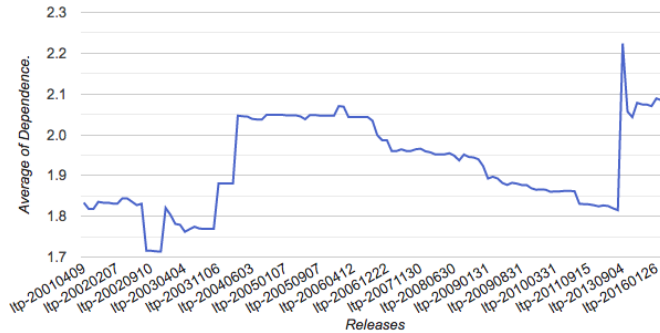


Figure 4.19: Average of Unit Dependence per Release.

Observation 6. *LTP has presented high values of cloned methods in the releases, and some of these releases present values higher than 50%. Additionally, the percentage of files associated with clones presented high values. These are good indicators that the maintenance and evolution of the project are not easy. Thus the detection, monitoring, and removal of code clones is an essential task in the future.*

4.3.2.4 M6 - Dependence: is measured as the number of unique outgoing calls (fan-out) from a test code unit to production code units [23]. In the context of test code, the modules' coupling is minimal [23], and high values indicate that the test can be divided into more units. Unit dependency affects the changeability and the stability of the test code. Changeability is affected because changes in a highly coupled test are harder to apply since all the dependencies to the production code have to be considered [101]. At the same time, stability is affected because changes in the production code can propagate more easily to the test code and cause tests to brake, increasing the test code's maintenance effort [23, 101].

Figure 4.19 shows the average of unit dependency metric per release. The unit dependency values presented a mix of sub-linear and super-linear growth with trending of growth. However, in the average, the LTP unit dependence presented low values (less than 3) considering the thresholds suggested by Athanasiou et al. [23] for test code. Moreover, we explored the unit dependence per subsystem, and **Network** presented the highest growth values. The **Kernel** has been growing slightly, while the **Commands** and **Misc** have stable average values during the evolution.

We also analyzed the distributions of unit dependency using an empirical distribution function (ECDF), as shown in Figure 4.20. Using this distribution, we can observe that 3.1% of the test programs are classified as having moderate values ($3 < unitdependency \leq 4$) and 4.5% are classified as having high values (unit dependency between 4 and 6). Moreover, 3% of the test programs presented very high values (unit dependency greater than 6). While the results for test programs with low unit dependency values are encouraging, one should also consider the high values of unit dependence in the tail of the distribution. According to our results, 4-7% of the test programs have a unit dependency

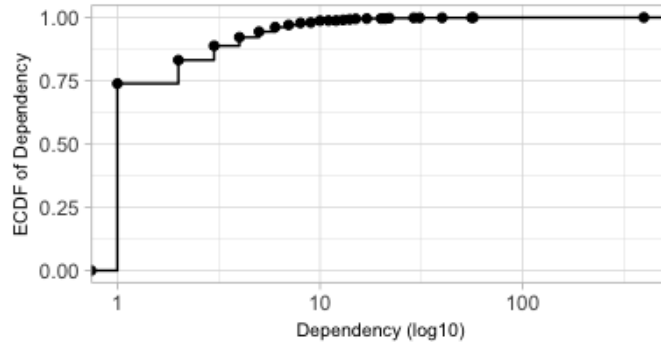


Figure 4.20: Percentiles of Unit Dependence.

above 4, classified by Athanasiou et al. [23, 101] as 'complex to maintain'. The top values observed are extremely high: for example, in the release `ltp-20130904` we have a test program (`prot_hsymlinks.c`) with unit dependence of 396; in the release `ltp-20021210` there is a test program (`sysconf01.c`) with 57. Such values are totally out of the scale.

Observation 7. *In general, LTP presented low values of unit dependence, but some test programs present values extremely high (unit dependency > 6) and deserve special attention. At the level of subsystems, the **Network** presents the highest growth values, while the **Commands** and **Misc** have stable average values during the evolution.*

4.3.3 RQ3 - How does the test suite change?

The evolution of HCS is the rule and not the exception due to ever-changing requirements and the long-term use of many systems [5]. In the same way, tests need to change to fit the new requirements [40]. To evaluate the maintainability related to changes in the project, we extracted values for two metrics. These metrics analyzed the basic operations (addition and exclusion) performed in the test cases and code changes performed in the test suites. In the sequence, we present a descriptive statistical view of these metrics during the evolution process.

4.3.3.1 M7 - Test Case Changes: LTP presented changes as a result of evolution. We identified that new test cases are introduced, removed, split, merged and renamed. These evolution changes are expressed by two basic operations: test cases added and removed. By processing the LTP test cases history, we extracted a dataset of changes linking them to their specific release, as can be seen in Figure 4.21. As could be expected, the absolute numbers tend to grow with time. The number of test cases removed presented a stable behavior with modest growth at the end of the series. However, the number of test cases added seems to be relatively unstable. Additionally, we can observe that, despite the size and complexity of LTP, adding and removing test cases are performed with a regular frequency.

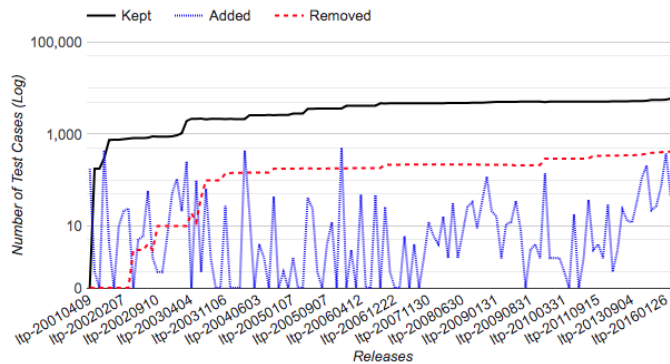


Figure 4.21: LPT evolution from the test cases change point-of-view.

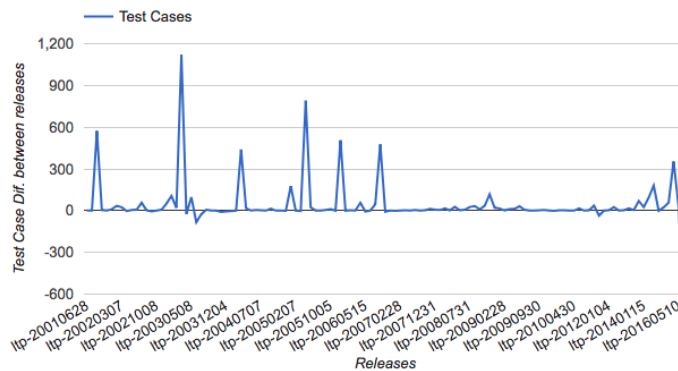


Figure 4.22: Changes in the number of Test Cases between releases.

Figure 4.22 shows the difference in the number of test cases among two releases *i.e.*, considering the total of test cases (tc) in a release (n), the difference (d) is: $d = tc(n) - tc(n-1)$. We can observe that the growth rate is very irregular. Moreover, every relatively large growth is always followed by a drop, resulting in an alternation between growth and stabilization. Also, we explored in depth the data in some hills show in Figure 4.22. For example, in release `ltp-20030306`, we observed that more than 800 test cases were added as a result of the introduction of stress tests to LTP. On the other hand, we can highlight a significant decrease in the two last releases. This can be traced to the exclusion of test cases in some test suites and the exclusion of 5 test suites, resulting in a total of 532 test cases excluded. These exclusions were performed mainly in test suites related to the network subsystem, such as: `network.stress`, `network.commands`, and `ipv6`.

Additionally, we analyzed how long a distinct test case is kept in LTP, as can be seen in Figure 4.23. In general, 23% of the test cases remain in a few releases (1-10), and 4.1% of the test cases stay on all or almost all releases (121-122). Examining the first set of test cases that stay between 1 and 11 releases, we can highlight that these test cases are mainly present in two test suites: `syscalls` and `containers`. This can be explained, since `syscalls` is a large and important module of the Linux Kernel resulting

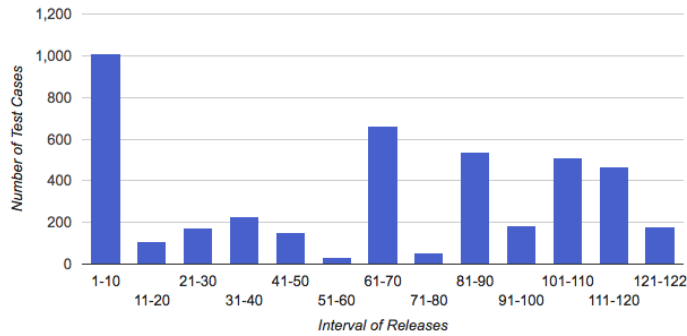


Figure 4.23: Number of Releases that a Test Case Remains in LTP.

in more exposure to changes. The `containers` test suite supports testing of the container functionality. This test suite contains test cases related to new functionalities: `utsname`, `PID Namespace`, and `user process ID` that remained through just 5 releases in LTP. Investigating test cases that are present in all or almost all releases, *i.e.*, 120-122, we found that these test cases are related to fundamental parts of the Linux Kernel. A significant portion (70%) of them is concentrated in the `quickhit` test suite. This suite is a small subset of `syscalls`, which performs tasks such as: create and move files to a temporary directory, handles some common command line parameters, clean up, and so on.

Observation 8. *The addition and exclusion of test cases in LTP are performed with regular frequency. Moreover, every relatively large growth is always followed by a drop, resulting in an alternation between growth and stabilization. In general, 23% of the test cases remain in a few releases (1-10), and 4.1% of the test cases stay on all or almost all releases (121-122).*

4.3.3.2 M8 - Test Program Changes: To evaluate code changes, we observed the difference among test programs (test code files) in the releases. This way, we have a detailed examination of the code changes from one release to the next, *e.g.*, if the code of one file in a release X changed to next release X+1. Figure 4.24 shows the number of test programs that had code changes in each release. In the same way that the metric M8, we can observe that the change rate is very irregular. Additionally, Figure 4.24 also shows jumps, such as: `ltp-20110228` and `ltp-20130109`. These hills can be explained by activities of clean-up/maintainability on the source code. For example, in release `ltp-20110228` not only big changes in program code were made, but also 1.199 code files and 152K lines were excluded.

Figure 4.25 shows the number of test program changes per subsystem. In this figure, we show the top subsystems in changes, not only the four subsystems presented from the initial releases. As we expected, the `kernel` subsystem concentrates significant code file changes, since this subsystem has more LOC and test cases. Besides, the vast majority of subsystems do not present changes among two consecutive releases. Figure 4.26

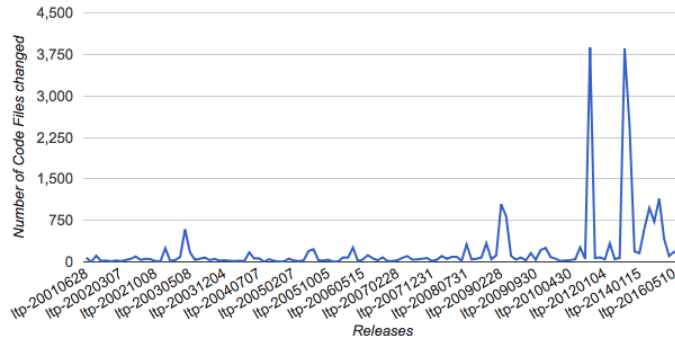


Figure 4.24: Number of changes in code files per releases.

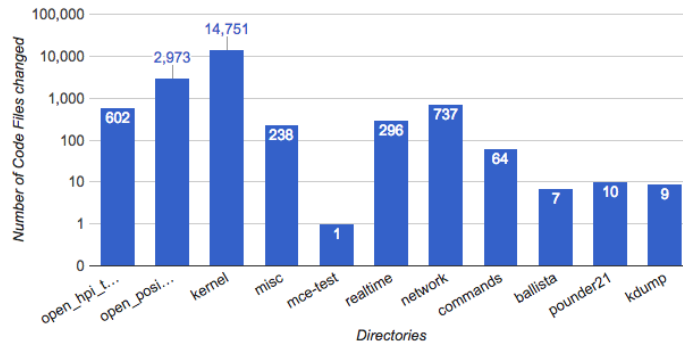


Figure 4.25: Test programs changed per subsystem.

shows the top eleven code files with more changes. Interestingly, many of these code files (1-1.c, 5-1.c, 2-1.c, 4-1.c, 9-1.c, 6-1.c and 8-1.c) belong to the OPEN POSIX subsystem. This subsystem contains test programs with the goal of performing conformance, functional, and stress testing in conforming the IEEE 1003.1-2001 System Interfaces specification. Other files, such as `swapon02.c` and `sendmsg01.c`, are from the `kernel` subsystem.

Observation 9. *Changes in code files are performed regularly, and the rate is very irregular. We note that some releases, such as `ltp-20110228` and `ltp-20130109`, present changes much higher than average, as a result of activities of integration or clean-up/maintainability on the source code. Moreover, code changes are mainly performed in the `kernel` subsystem, it can be related to the factor that this subsystem has more LOC and program files. Also, we observed that many of the top code files with more changes are in the OPEN POSIX Test Suite subsystem. It can be explained by the volatile nature of the files in this subsystem since they attended to international specifications that are updated with frequency.*

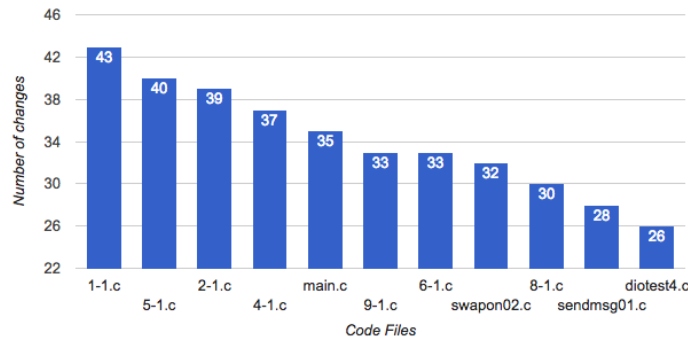


Figure 4.26: Test programs with more changes.

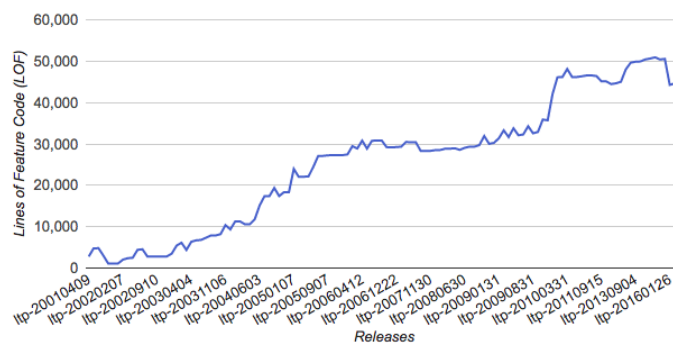


Figure 4.27: Lines of Feature Code evolution.

4.3.4 RQ4 - How does variability affect the maintainability of the test suite?

We used a set of variability metrics to evaluate LTP variability, and how the variability affect the maintainability of the test suite. These metrics have been discussed [89] and used [77, 91, 106, 107] for implemented artefacts in HCS. The findings related to each metric are discussed next:

4.3.4.1 M9 - Lines of Feature Code (LOF): This metric is an adaptation of the classic lines of code metrics [89]. This metric counts the number of lines that are linked to feature expressions, e.g., surrounded by `#ifdefs-blocks` [93, 88]. This way this metric indicates whether a small or a significant fraction of the code base is variable.

Figure 4.27 shows the number of LOF in the test programs summed up per release. We can observe that the LOF size has increased from 2 KLOC at the beginning to about 50 KLOC during LTP evolution. Discrete jumps in the growth curves *e.g.*, from `ltp-20091130` to `ltp-20091231` can be traced to the inclusion of a few test code files with high LOF values, such as the `linux_syscall_numbers.h` and the `commands.c` that respectively added 4404 and 932 new lines of features code in version `ltp-20091231`. The `linux_syscall_numbers.h` file contains all of the logic for Linux system call numbers. The `commands.c` file contains several commands necessary to perform memory opera-

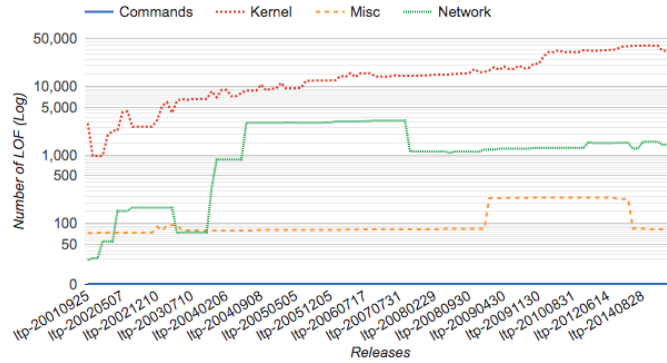


Figure 4.28: LOF in each subsystem.

tions, such as: mapping, protection, and faulting. This way, the implementation of these files are usually based on multiple `#ifdef` statements, resulting in the high LOF values observed. These high values of LOF could be a warning for tests that are difficult to understand, making them hard to maintain. Figure 4.28 shows the number of LOF over time per subsystems. The evolution behavior of LOF in some subsystems is similar to LOC evolution. For example, the `Network` and `Misc` subsystems presented expressive jumps and irregular behavior during evolution. Similarly to LOC, the `Kernel` subsystem presented consistent growth of LOF for the entire period and consequently demanding more effort for maintenance. Interestingly, the `command` subsystem did not have LOF in any releases.

According to Liebig et al. [93], the fraction of cpp-annotated code (LOF/LOC) in some mid-size software systems exceeds 50% of the code base. However, large software systems (Freebsd, GCC, Linux, and Opensolaris) contain a small percentage of variable source code compared to the average. Hunsen et al. [88] analyzed the variability in 20 open-source systems, and reported that these systems have an average of 22% of cpp-annotated lines of code. However, in big systems, the percentage of all lines of code annotated vary from 9% to 14%. In spite of we had analyzed test code, a different set of code from all the ones cited here, our results also support this. We observed that the amount of LOF represents high values (more than 20%) of the code base, in releases with small LOC size (1st, 2nd, and 3rd releases), as can be seen in Figure 4.29. However, there is a lower percentage of variable source code in large releases (range 1-8%). Although large releases present a small portion of variable source code, we cannot affirm that these releases are more comfortable to maintain since the absolute values of LOF are big, and it can result in more feature interactions [108].

Observation 10. *The LOF metric shows an increase in absolute values during evolution. We observed that some test cases present high values of LOF, and they are difficult to understand. At the level of subsystems, the `Kernel` shows the consistent growth of LOF for the entire period. Interestingly, the `command` subsystem did not have LOF in any release.*

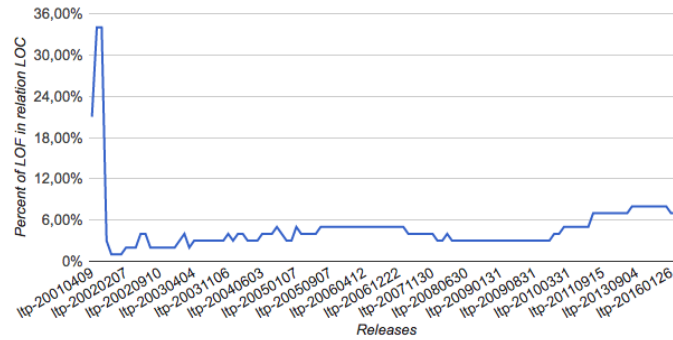


Figure 4.29: Fraction of cpp-annotated code (LOF/LOC) per Release.

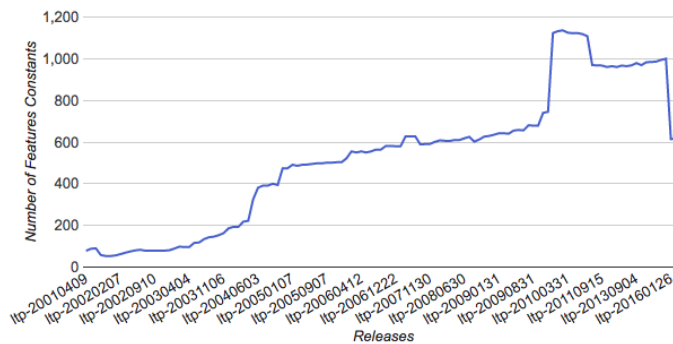


Figure 4.30: Number of Features Constants (NOFC).

4.3.4.2 M10 - Number of Feature Constants (NOFC): This metric counts the number of feature constants used inside of code artifacts, *e.g.*, distinct feature constants in `#ifdef` blocks [89]. The NOFC metric reflects directly the configuration dimension and provides important insights regarding to variability and complexity [93].

The NOFC metric value is increasing over time as can be seen in Figure 4.30. Likewise, the results obtained by Liebig et al. [93], the variability of the project increased with its size, *i.e.*, the NOFC and LOF metrics increased as a result of LOC growth. Additionally, we can observe that the NOFC growth rates can be seen to be super-linear up to version `ltp-20100131`, but it exhibits a significant decrease in the last year. Analyzing the data more closely, this behavior is associated with LOC. If we compare the same period of Figure 4.7, we can observe that a lot of lines of code were removed, as a result of test cases reduction (Figure 4.21), reflecting directly in the configuration dimension. Figure 4.31 shows the number of NOFC over time per subsystems. The `Kernel` and `Network` present consistent growth of NOFC for the entire period and consequently demanding more effort for maintenance while the `Misc` subtly increased. `Command` subsystem did not have NOFC values since the specific nature of this subsystem make the variability does not seem necessary.

Figure 4.32 shows that the NOFC metric presents the higher values from the `ltp-20091231` release to `ltp-20101031`. In the same period, few test cases were added, as can be seen

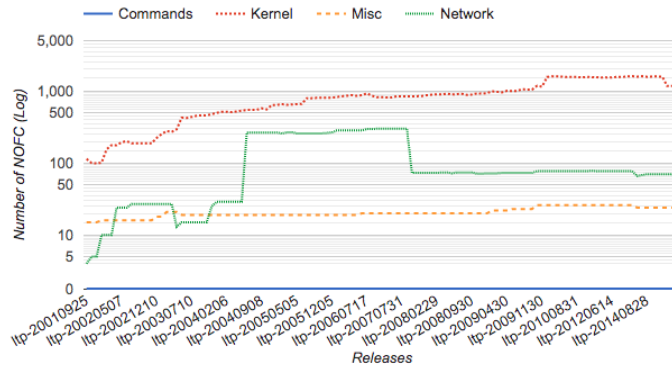


Figure 4.31: NOFC in each subsystem.

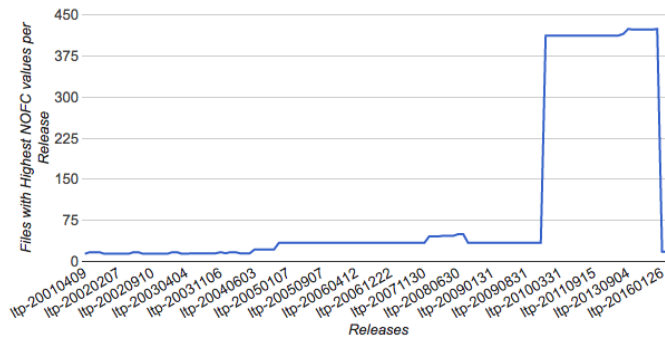


Figure 4.32: Highest NOFC values per Release.

in Figure 4.22. The high values of NOFC in this period can be traced to the addition of new test cases with more feature constants than the average. In fact, as can be seen in Figure 4.32, the hill from `ltp-20091231` release to `ltp-20150903` is mainly the result of adding just one new file (`linux_syscall_numbers.h`). This file added 412 new feature constants, and its value is above the average of 0.5 per file. The extremely high NOFC value observed in the `linux_syscall_numbers.h` file raises the question about what is this file? The file, from `kernel` subsystem, contains all logic for Linux system calls and the implementation is usually based on multiple `#ifdef` statements with a `#define` directive in each one leading to the extreme NOFC values observed.

Observation 11. *The data revealed that the variability (feature constants) increases with the size of the project, resulting in more complexity as the project grows and consequently making it harder to maintain. Additionally, few test cases had contributed to increasing variability, and deserve special attention. We note that `Kernel` and `Network` present consistent growth of NOFC for the entire period, while the `command` subsystem did not have NOFC values since the specific nature of this subsystem make the variability does not seem necessary.*

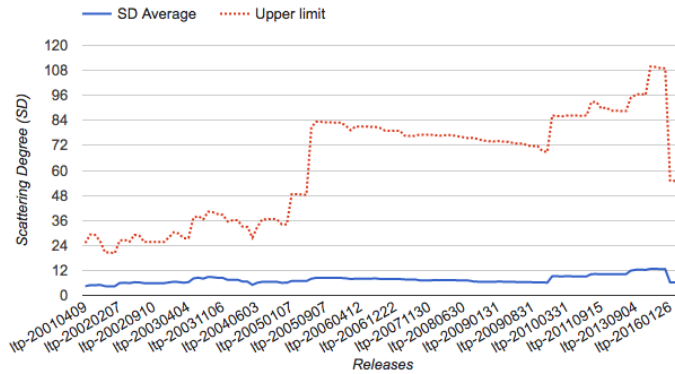


Figure 4.33: SD Evolution.

4.3.4.3 M11 - Scattering Degree (SD): The SD metric indicates the number of feature constants occurrences in different feature expressions [93], *e.g.*, in how many files a feature constant is used in [89]. According to Queiroz et al. [106], SD is long said to be an undesirable characteristic of source code. The relation between SD and maintainability is intrinsic since it introduces extensions across the code base and their maintenance requires to analyze and change different code locations, that may cause a ripple effect.

Figure 4.33 shows the SD average in an interval with a safety margin (95%). It presents a mix of sub-linear and super-linear growth with lower values at the initial series. The SD average increased over time with a substantial decrease in the end. Additionally, we checked the the distributions of SD and their evolution of percentiles, as can be seen in Figure 4.34. Using a threshold suggested by Zhang et al. [107] we can observe that 50% of the test programs present SD values less than 2, i.e, easy to maintain. We identified that the SD metric has the highest values between the versions `ltp-20091231` and `ltp-20150903`. Notably, the standard deviation is quite high in these releases. Thus, a significant number of feature constants results in a high SD and the respective implementation scatters possibly across the entire release.

To study the feature constants with high-SD, we focused on the tail of the distribution. The fraction of feature constants with high SD is small and even decreasing. We observed that in most parts of releases less than 25% have an SD over 2, and less than 10% have an SD over 6. The data shows that the LTP allows most features to be incorporated without causing any scattering. However, some features do not fit well into the architectural model and are scattered across the source code. Moreover, the proportion of scattered features is nearly constant, which may indicate that it is an evolution parameter actively controlled throughout the LTP evolution. We identified that the highest SD feature constant in the whole dataset is `_sparc_` in release `ltp-20150119`. As shown in Figure 4.35, the feature constant was introduced to the LTP only in release `ltp-20061007`. It has grown since then reaching an SD value of almost 652 at the end of 2014. However, in release `ltp-20160126` the SD of this feature constant dropped drastically to the SD value of 7.

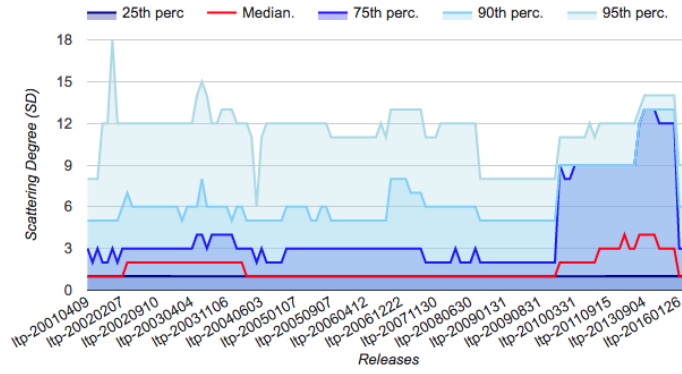


Figure 4.34: Percentiles of SD per Release.

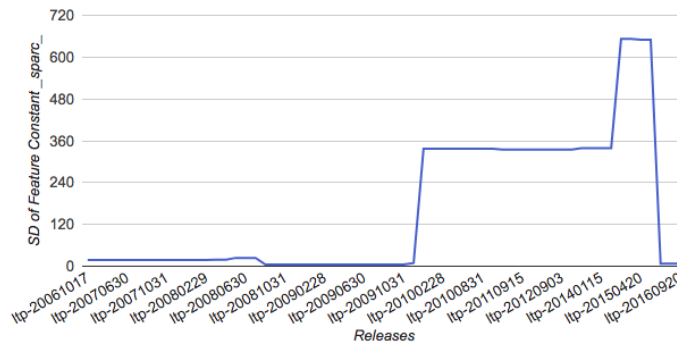


Figure 4.35: Evolution of the `_sparc_` feature constant.

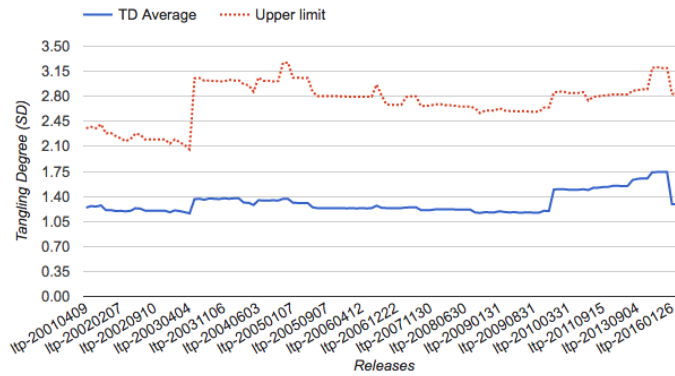


Figure 4.36: Tangling Degree Evolution.

Observation 12. *Many feature constants in LTP presented a low scattering level. However, there are a few feature constants with a high scattering degree that deserve special attention by the development team since they could require a considerable maintenance effort after changes.*

4.3.4.4 M12 - Tangling Degree (TD): The TD metric serves as a counterpart to the scattering degree metric [89]. Instead of measuring how many elements are associated with a specific feature constant, it indicates the number of different feature constants that occur in a feature expression. It should indicate the complexity in terms of variability realization by measuring the number of involved feature constants in an expression [89].

The TD metric presents a mix of sub-linear and super-linear growth, as can be seen in Figure 4.36. We observed that the LTP feature expressions presents low complexity, *e.g.*, $TD = 1$ on average [93, 88]. Low complexity is preferable because feature expressions that consist of a high number of feature constants impair program comprehension and difficult the maintenance. Additionally, we analyzed the distribution of TD and their evolution of percentiles (Figure 4.37). We observed that there are few feature expressions with a high tangling degree. In general, 25% of features expressions have TD up to 2. Figure 4.38 shows feature expressions with high TD. In this case, we focus on expressions with TD higher than 5, which was suggested by Zhang et al. [107] as indicating expressions complicated to understand. The fraction of feature expressions with high TD is small, and the absolute number of such feature expressions did not grow representatively with time.

Observation 13. *The TD metric presents low values during the LTP evolution, resulting in a low complexity of feature expressions. However, there are a few feature expressions with a high tangling degree, that could impact negatively in the program comprehension and maintainability.*

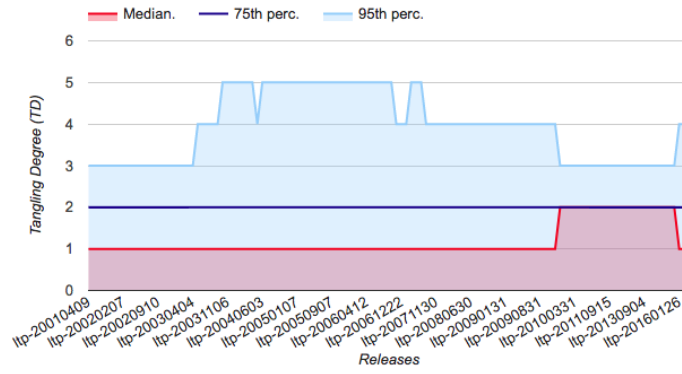


Figure 4.37: Percentiles of TD per Release.

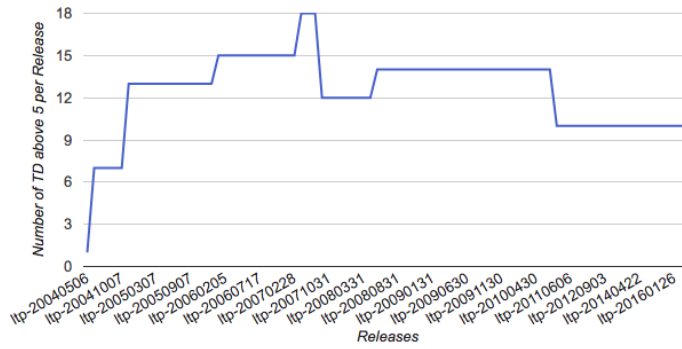


Figure 4.38: Feature expressions with high TD evolution.

Table 4.1: Summary of Findings.

Research Question	Metrics	Behavior and Mean Values									
		General		Kernel		Network		Misc		Commands	
RQ1	M1-Test Cases	↗	366.5	↗	702.2	↗	111.0	↗	28.8	→	12.6
	M3-Contributors	↗	2.9	↗	2.5	→	1.4	→	1.1	→	1.2
RQ1 and RQ2	M2-Unit Size	↗	185.8	↗	290.9	↗	172.1	↗	210.0	→	11.6
RQ2	M4 - Complexity	↗	5.3	↗	5.5	↘	5.5	→	4.6	→	1.3
	M5-Duplication	↗	20%	-	-	-	-	-	-	-	-
	M6-Dependence	↗	1.9	↗	1.6	↗	6.8	→	0.9	→	0.5
RQ3	M7-Test Case Chg.	↘	225.7	-	-	-	-	-	-	-	-
	M8-Test Prog. Chg.	↗	204	↗	124	↗	12	↗	7.2	↗	6.4
RQ4	M9 - LOF	↗	7.1	↗	13.8	↗	7.9	↗	2.4	→	0
	M10 - NOFC	↗	0.46	↗	0.72	↗	0.62	↗	0.42	→	0
	M11 - SD	↗	4.9	-	-	-	-	-	-	-	-
	M12 - TD	↗	1.7	-	-	-	-	-	-	-	-

4.4 DISCUSSION

In this section, we examine the collected data providing answers for each of our RQs. Table 4.1 shows a summary of the results. The findings were classified based on different categories. Firstly, they were classified according to the research question and subdivided in each metric. Then, we have the behavior of evolution for LTP (General) and the main subsystems analyzed (Kernel, Network, Misc, and Commands), which exposes: if the evolution behavior of the metrics is increasing (↗), decreasing (↘), or stable (→). Additionally, the mean values are presented next to the evolution behavior. The mean values highlighted in red represent metrics that present values higher than the thresholds suggested in the literature for accepted values. Note that some values in the table are filled with (-), and we have two main reasons for it: (i) the values are not applicable to the metric, and (ii) the values are not available per subsystem. The second reason occurs since we used a set of scripts and third-party tools³ to extract the information, and some of them just provide the results by releases.

In summary, the effort to develop the test suite is high. By analyzing the metrics related to RQ1 and tabulating their trends over time, we observed that LTP presents continuous growth in unit size (LOC), functional content (Test Cases), and contributors. As a result there is no indication that its adoption rate is decreasing, and probably more effort will be necessary to evolve the project in the future. We notice that the unit size

³NiCad [105] - <https://www.txl.ca/txl-nicadownload.html>, Testwell CMT++ - <http://www.testwell.fi/cmtdesc.html>, and cppstats [93] - <http://fosd.de/cppstats/>

has increased significantly from the initial to the current release. Although the number of contributors had increased too, when we consider the ratio between LOC and number of unique contributors per month, the LTP exhibits values incredibly high, indicating that the project contributors had a great deal of effort to develop and to evolve the test suite. Additionally, we observed that considerable effort was spent in the early releases when the project increased fast as a result of the initial implementation. This way, the development teams that intend to start developing tests have to consider allocating more contributors at the beginning of the project. At the level of subsystems, the `Kernel` presents the most consistent LOC growth and contains considerable part of the test programs. This way, this subsystem has deserved special attention (number of unique contributors), and it concentrates great part of the effort of development for the whole period.

Although not all metrics selected to evaluate the maintainability present unsatisfactory indicators, the overall assessment of test code maintainability is low. We observed that the LTP's maintenance is not easy and could become more difficult as the project evolves. Even though the unit size growth curves seem to be slowing down since 2011, a significant percentage (58%) of the test programs presented very high values (unit size greater than 48), and they are susceptible to the occurrence of test smells [101]. Additionally, the project presents high values of cloned methods (56%), and some of the releases show values higher than 50% of the duplicated code. Thus, the detection, monitoring, and removal of code clones is essential in the future. Another factor contributing to the poor indicator of maintenance involves a large number of functions (56%) that present very high unit complexity values. Additionally, we found no improvement of the functions with top values of unit complexity from the first to the last release. Regarding subsystems, the `Commands` subsystem has slightly increased the unit complexity average in the entire period. The improvements of the unit complexity values in the last years are linked with `Kernel` and `Network` subsystems. On the other hand, LTP presents low values for unit dependence, which is a good indicator of test code stability regarding changes in the production code.

We identified that new test cases are introduced, removed, split, merged and renamed. Two primary operations express these evolution changes: test case addition (introduction, division, and the junction of test cases), and exclusion (removal of test cases). We observed that despite the size and complexity of the LTP, the addition and exclusion of test cases are frequent. A high changeability is a fact that may significantly increase the test maintenance cost. It also affects stability, since the changes can introduce logical errors in the test code. After we analyze the difference in the number of test cases among two releases, we observed that the growth rate is very irregular. Every relatively large growth is always followed by a drop, resulting in an alternation between growth and stabilization. Additionally, we noticed that the test case life cycle is short, and it is not a good indicator of low maintainability [23]. At the level of test programs (code files), we note that test programs with variability remain less time than test programs without variability. This can be an indicator that the variability influence negatively in the maintainability since it introduces more changes.

According to Lehman [109], it is not simple to distinguish the changes between adaptation to the environment and general growth. However, some LTP changes are easily

identifiable, since the test cases are grouped in test suites. For example, the addition of 506 new test cases in the release `ltp-200501003` is related to changes in the hardware environment in the network category. Also, we observed that code changes are mainly performed in the `kernel` subsystem, it can be related to this subsystem has more LOC and program files, consequently, more actions are presented in the artifacts inside this subsystem.

We observed that variability impacts mainly in the complexity and quality of the test maintainability. Additionally, we identified that LOF and NOFC metrics deserve special attention by the development team since they provide good indicators about the maintainability in the test suite. These metrics have a direct correlation with LOC, *i.e.*, as the LOC increases, they also increase, and as a consequence, the maintainability decreases. Moreover, we identified that some test cases present high values of LOF, and they are difficult to understand. Interestingly, one file (`linux.syscall_numbers.h`) had contributed to increase in a considerable way the NOFC values, and the LTP contributors need to be watchful to this file. We noticed that `Kernel` and `Network` presented consistent growth of LOF and NOFC for the entire period. Interestingly, the `command` subsystem does not present values for variability metrics since the specific nature of this subsystem seems to make unnecessary the variability. On the other hand, many feature constants in LTP presented a low scattering level. Low feature scattering is a preferable situation [106], and it is a good indicator for maintainability of the project. Additionally, the tangling degree metric presented low values. This factor contributes to the low complexity of feature expressions, and helps the maintainability.

4.5 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study. To improve the validity of this work, each step was carefully performed; however, there are threats to the validity of the study, which are briefly discussed.

4.5.1 External Validity

- **Study object:** Our study is based on a single project (LTP). However, the nature and size of LTP (122 releases, 400,000 test cases, 16 years of development, and 63 million lines of code) make it a complex, representative case of software test evolution with variability.
- **Feature detection:** Our analysis is limited to the test code and `#ifdefs`. Nevertheless, representation and implementation of features are not limited to `#ifdefs` [106]. Additional configuration layers, such as configuration scripts or tools could be used. A comprehensive analysis of these configuration layers was out of the scope of the study.

4.5.2 Construct Validity

- **Selected metrics:** Metrics of size, complexity, and variability were extracted in this study, and inappropriate metrics might have been chosen. This threat was mitigated by choosing the most used metrics, such as Lines of Code (LOC), McCabe Cyclomatic Complexity [102], and variability metrics [93, 94, 88]. Moreover, they are widely known and have been used in other studies analyzing HCS [99, 93, 88].

4.6 CHAPTER SUMMARY

The benefits of having a good test suite are extensively studied in the literature [5, 110, 41]. In the context of configurable systems, prior studies have discussed the importance of tests during the development process [16, 111, 11, 10, 40]. However, only a few studies examine aspects related to test evolution in configurable systems [112]. Therefore, we mined the versioned history of the Linux Test Project, a functional and regression test suite for testing the Linux Kernel. In particular, we studied the test suite evolution with special focus on the effort, maintainability, changes, and variability aspects. A set of thirteen observations was extracted providing answers for the research questions. Based on these observations, we can conclude that the effort to develop a test suite for a large configurable system is high. We identified that the LTP presents continuous growth in number of test cases, resulting in more effort as the project evolve. We note that special attention needs to be reserved at the beginning of the projects since at this time they have not many contributors and tend to grow fast increasing the workload of development team. Additionally, we identified that the test code maintainability is low, and as the project evolves the unit size, complexity, and duplication increase, contributing for a hard maintenance.

We identified that the addition and exclusion of test cases are performed at a regular frequency. As a result, it could increase significantly the effort to maintain the test suite. Additionally, test programs with variability remain less time than test programs without variability. It can be an indicator that the variability influence negatively in the maintainability since it introduces more changes. We identified that LOF and NOFC metrics have a direct correlation with LOC, and their growth decrease the maintainability of the test suite. Also, we identified the development teams need to be watchful to some feature constants with a high scattering degree since they would require a considerable maintenance effort in case of changes.

This research is the first attempt to analyze test evolution evaluating characteristics that indicate the effort to develop and maintain the test suite, and considering variability aspects. Moreover, we provide static analysis of the test suite and make available a dataset that allows other researchers to investigate test evolution, co-evolution, and dynamic analysis in future studies. We believe that our results and empirical observations contribute to the advance of the research concerning test evolution in configurable systems.

The next chapter presents our study of test evolution in 36 open-source projects from various domains. We aim to get more evidence to understand test evolution in configurable

systems and its similarities and differences to Single Systems (SS). The similarities will help to adopt techniques from SS to configurable systems and vice versa. Additionally, the differences will help us to design new techniques addressing the difference.

TEST EVOLUTION IN CONFIGURABLE SYSTEMS AND SINGLE SYSTEMS: A COMPARATIVE STUDY

In the previous chapter, we realized a case study in a large configurable system (Chapter 4). We presented a set of observations and implications for open-source teams and practitioners. However, we have observed, among other issues (Chapter 3), the lack of research into test evolution in reusable systems combining evidence from different sources. Thus, additional systematic reports are necessary to provide more evidence about the test evolution in reusable systems. In this context, this chapter presents a comparative study to evaluate the test evolution in 18 open-source projects from various sizes and domains in configurable systems and its similarities and differences to 18 SS projects. The comparison allows us to identify similarities that can help us adapt techniques from SS to configurable systems and vice versa. The differences can help us design new strategies addressing the discrepancies.

A better understanding of this evolution can allow us, in the medium-term, to come up with prediction models, guidelines, and best practices that will enable the community to improve their current practices, tools and to make the software tests less expensive and time-consuming. Thus, we evaluate the test evolution in 36 open-source projects of different sizes and from various domains. Additionally, we analyze more than 7 millions of lines of code and 1.3 millions lines of test code in total.

The remainder of the chapter is organized as follows: Section 5.1 presents the methodology and data collection process; Section 5.2 describes the results of the study and presents a set of observations about the effort of development, maintainability, and differences in the test evolution between SS and HCS; Section 5.3 presents the discussion and implications of our results; and Section 5.4 presents threats to validity.

5.1 METHODOLOGY

This section explains our methodology and data process.

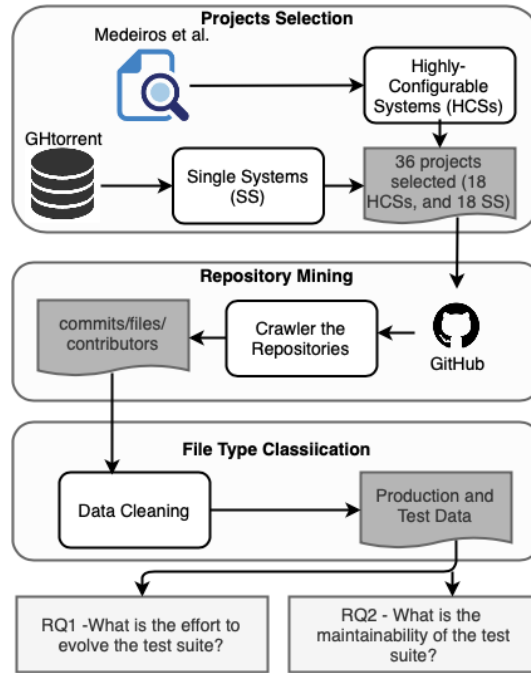


Figure 5.1: Overview of our subjects selection and data collection approach.

5.1.1 Study Subjects

In order to investigate test evolution, we selected a representative set of projects. We used two different approaches to select the projects, as can be seen in the first part of the Figure 6.1. We choose the HCS projects from the list of 63 C/C++ consolidated HCS projects used by Medeiros et al. [53]. We identified 18 HCS projects that had a test folder in its structure with at least ten test cases by manually identifying the test directory. A different approach was used to choose SS projects. Using GHTorrent¹, we selected the 18 top C public projects with highest number of stars on GitHub². According to Borges et al. [113], number of stars in a project are viewed by practitioners as the most useful measure of popularity at GitHub. Our final list contains projects of different sizes and from various domains, such as games, operating systems, Web servers, database systems, and so on. Summary of the selected HCS and SS projects can be seen in the Tables 5.1 and 5.2 respectively.

5.1.2 Data Collection

Two main steps were performed for the data collection process (second and third part in Figure 6.1):

- *Repository Mining*: We downloaded the git repositories for these projects from

¹<http://gtorrent.org>

²<https://github.com> - the world's most extensive collection of open source software, with more than 28 million users and 79 million repositories [113].

Github to collect various information such as: who authored the code, when the code was committed, the name of the file involved, and so on. In total, we collected 585,632 commits from all the projects, for the project start date until January 31st, 2019.

- *File type classification*: Since our goal was to analyze test evolution, we classified each unique file in the analyzed projects as either System Under Test (SUT) file (*i.e.*, files that contain code related to system functionality), or test file (*i.e.*, files that contain code related to tests). Those files that do not fit in any category are marked as “other”, such as release notes, manuals, etc.

5.1.3 Data Preparation

As we aim to investigate how the test suite evolves, we could use different ways of partitioning time. Some researchers [114, 115] have used releases as the unit of time, other individual commits, or discrete-time units (years, months, weeks, days) [116, 19]. The same way that [25, 19], we selected the month as our unit of measure because, while subject to some variation from project to project, it gives us enough details into the evolution of projects. Additionally, by checking the distribution of commits across the analyzed projects history, we found that these projects had an active history of 102 months (median) after they started to have tests. This way, we cut off analysis at 102 months to not skew our findings.

5.1.4 Tools Selection

We evaluated some aspects and metrics to answer each research question. So different tools were adopted to extract the information needed. In **RQ1**, three aspects were evaluated: contributors, modified files, and assertions. For measurement the number of contributors and modified files, we used the commits obtained by mining the repository. For assertions, we ran a custom script on each snapshot (last commit in each month) of the projects. The output of this process included information, such as number of assertions, and assertion density. In **RQ2**, four metrics were analyzed: unit size, unit complexity, unit dependence, and duplication. To measure the unit size and dependence, we used *scitools Understand*³. This tool offers metrics reports, and its reverse engineering feature provides a complete structure of the code dependence. Although the *scitools Understand* provides measurement for unit complexity, it does not consider the variability aspects on code in the computation of complexity. Thus, we used *MetricHaven*⁴ [117] for complexity. This tool supports McCabe’s cyclomatic complexity as well as the variability aspects on the code [89]. Finally, in the duplication analysis, we used *NiCad*⁵ clone detector [105]. This tool enabled us to find exact and near-miss clones.

³<https://scitools.com/>

⁴<https://github.com/KernelHaven/MetricHaven>

⁵NiCad - <https://www.txl.ca/txl-nicaddownload.html>

Table 5.1: HCS Projects Selected.

Highly-Configurable Systems							
Project	Stars	Contributors	LOC	tLOC	Start Date	Start Test	Domain
<i>Angband</i>	613	48	114203	4768	2007-03-23	2007-08-09	game
<i>Clamav</i>	784	24	1277880	4856	2003-07-28	2008-03-13	antivirus
<i>Curl</i>	12734	497	192088	49625	2000-01-10	2000-11-10	data transferring tool
<i>Gcc</i>	2598	610	8365418	1717860	1988-11-23	1997-08-19	compiler
<i>Irssi</i>	1954	85	67624	665	1999-09-03	2017-12-07	chat client
<i>Kerberos</i>	233	66	380797	26432	1987-10-23	1990-02-03	network authentication protocol
<i>Libsoup</i>	35	193	60230	18280	2000-12-07	2002-11-11	SOUP library
<i>Libssh</i>	107	8	30575	2606	2005-07-05	2008-03-17	SSH library
<i>MPSolve</i>	13	3	34352	3085	2001-12-31	2001-12-31	mathematical software
<i>Opensc</i>	1089	120	173774	5069	2001-10-19	2001-10-19	smart card tools
<i>OpenSSL</i>	9998	392	570966	84352	1998-12-21	1998-12-21	Internet Protocol
<i>Opentx</i>	919	78	600524	2834	2011-09-13	2014-06-30	radio transmitter firmware
<i>Openvpn</i>	3579	101	88809	1386	2005-09-26	2012-03-22	virtual network tool
<i>Ossec-hids</i>	2578	124	136329	2082	2005-09-23	2014-03-10	intrusion detection system
<i>Sleuthkit</i>	1334	66	377321	1584	2008-09-29	2009-05-08	command line tools
<i>Syslog-ng</i>	1067	92	146648	9021	2007-04-13	2007-04-13	log management
<i>Wiredtiger</i>	1336	27	170515	55598	2008-11-20	2009-05-27	data management platform
<i>xorg-server</i>	3	339	420576	7237	1999-12-30	2009-04-28	x server
Total	40974	2873	13208629	1997340			

Table 5.2: SS Projects Selected.

Project	Stars	Contribu- tors	Single Systems			Domain
			LOC	Start Date	Start Test	
<i>Arduino</i>	3381	320	129045	2014-11-15	2015-07-23	core for Arduino
<i>Ccv</i>	4363	9	238157	2010-02-06	2010-02-09	Computer Vision Lib.
<i>Cinder</i>	2921	102	810687	2010-04-21	2010-04-22	lib. for C++
<i>Grpc</i>	6869	447	635798	2014-11-27	2014-11-27	RPC framework
<i>Em- scripten</i>	7312	395	1374396	2010-08-26	2010-08-26	compiler infrastructure
<i>FFmpeg</i>	4603	1018	1179882	2000-12-20	2002-05-18	Multimedia Libraries
<i>Godot</i>	2843	837	1842512	2013-04-10	2017-01-11	Multi-platform game engine
<i>H2o</i>	5986	94	305545	2014-09-04	2014-09-04	HTTP server
<i>Libgit2</i>	3459	360	187720	2008-10-31	2008-11-02	cross-platform
<i>Libsodium</i>	6784	81	55782	2013-01-20	2013-01-20	crypto library
<i>Libui</i>	3808	30	43667	2015-04-06	2015-04-22	GUI library
<i>Libuv</i>	4547	321	65685	2011-03-23	2011-03-28	multi-platform support library
<i>Mpv</i>	3937	261	145141	2001-02-24	2001-02-24	Video player
<i>Netdata</i>	13845	251	170152	2013-06-17	2015-03-16	performance monitoring
<i>Obs-studio</i>	3828	247	340616	2013-10-01	2013-10-02	live streaming and screen recording
<i>Stb</i>	4112	131	69661	2014-05-25	2014-05-25	public domain libraries
<i>Toxcore</i>	8542	160	33823	2013-06-24	2013-06-24	communication platform
<i>Yoga</i>	11639	147	92803	2014-03-31	2014-10-07	cross-platform layout engine
Total	102779	5211	7721072			

Table 5.3: Wilcoxon Rank Sum test (p -value) and Cliff’s Delta (d) for SS vs. HCS projects.

Metrics	p -value	d
Contributors	3.7e-13	-0.173 (small)
Man-Month (M/M)	3.7e-13	-0.163 (small)
Modified Files	6.4e-14	-0.181 (small)
Number of Assertions	2.2e-16	0.015 (negligible)
Assertion Density	2.2e-16	-0.242 (small)
Unit Size	2.2e-16	-0.598 (large)
Unit Complexity	2.2e-16	-0.358 (medium)
Unit Dependence	2.2e-16	-0.414 (medium)
Duplication	2.2e-16	0.078 (negligible)

5.2 RESULTS

For each RQ, we discuss the metrics we used to address the RQ and our findings.

5.2.1 RQ1- How much effort is required to evolve test suite?

Our goal is to investigate the effort required to evolve tests in HCS. For this, we used maintainers’ effort metric suggested by Zhou et al. [25]. Zhou et al. identified that the number of contributors and number of modified files indicate the required effort. We also use the average number of assertions in test files as another metric as other researchers have used this as a measurement of effort [118, 23]. Detailed results regarding each of these metrics are provided in the following sections.

5.2.1.1 Number of Contributors: We analyzed the number of contributors involved in test activities during the evolution of the projects. Since different projects may vary in size, number of files and number of contributors, normalization is required for fair comparison across projects. We normalized by dividing the number of unique contributors in test files with the total number of unique contributors in each month.

Additionally, we tested if the number of unique contributors in test files is different between HCS and SS using the Wilcoxon Rank Sum test ($\alpha = 0.05$) [119]. Our results show that the difference is statistically significant between HCS and SS in terms of number of unique contributors contributing in test files. We also assessed the magnitude of the difference with the Cliff’s Delta (d) estimator [120]. To classify the effect size, we used the following thresholds [121]: *negligible* for $|d| \leq 0.147$, *small* for $0.147 < |d| \leq 0.33$, *medium* for $0.33 < |d| \leq 0.474$, and *large* otherwise. Our results show that the difference in effect size between HCS and SS is small ($|d| = -0.173$). Table 5.3 shows the result.

We observed that the evolution behavior of contributors is irregular for both types of projects, as can be seen in Figure 5.2. Also, the percentage of contributors involved in test activities at the beginning of the projects is the similar (32%) for SS and HCS. However, this value increases over time in HCS, reaching 57% at the end of the series. On the other hand, SS present almost the same amount 33%.

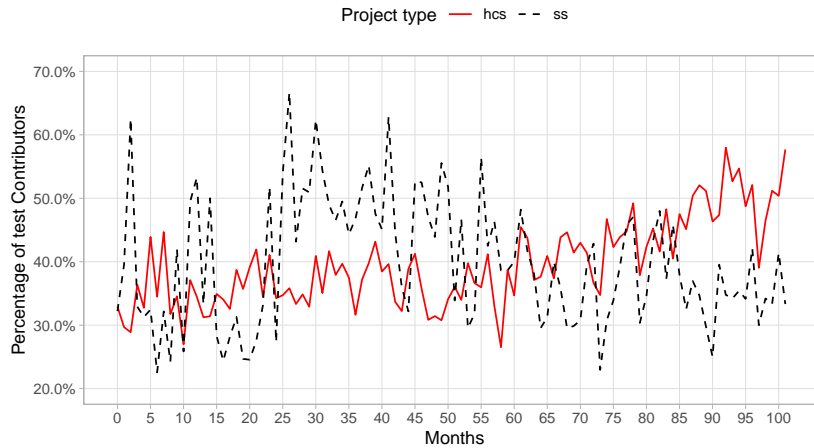


Figure 5.2: Contributors working in test files over time.

HCS and SS start with similar number of contributors but overtime more contributors start contributing to test activities in HCS.

5.2.1.2 Man-Month (M/M): We also investigated the workload of the contributors involved with test activities over time. We used Man-Month (M/M) to measure workload [100]. We extracted the number of test lines of code (tLOC) modified in a month and divide by unique contributors to measure Man-Month. Figure 5.3 shows the result. Please note that a logarithmic (log) scale was used in the vertical axis. This helps to show the growth curve clearly since the average values of man-month productivity present significant variations during the evolution. Man-Month productivity behavior is irregular for both types of projects. On average SS projects have higher workload compared to HCS. SS contributors write 1,509 *tLOC* per month on average, which is higher than HCS contributors writing 443 *tLOC*. SS contributors produce almost 400% more *tLOC* in a month than HCS contributors. Additionally, we tested if the Man-Month (M/M) in test files is different between HCS and SS using the Wilcoxon Rank Sum test ($\alpha = 0.05$) [119]. Our results show that the difference is statistically significant between HCS and SS in terms of Man-Month (M/M), but the effect size is small ($|d| = 0.163$).

SS contributors are almost four times more productive than HCS contributors.

5.2.1.3 Modified files: Zhou et al. [25] showed that more files a contributor oversees, more time and effort he will need to devote [25]. Moreover, tests can become invalid even when the changes preserve the behavior of production code [5] requiring more effort from the contributor side. Hence, we used number of test files changed in each month per developer as measurement of effort. Figure 5.4 shows the result. We ran Wilcoxon Rank Sum test ($\alpha = 0.05$) [119] and found a statistically significant difference between HCS

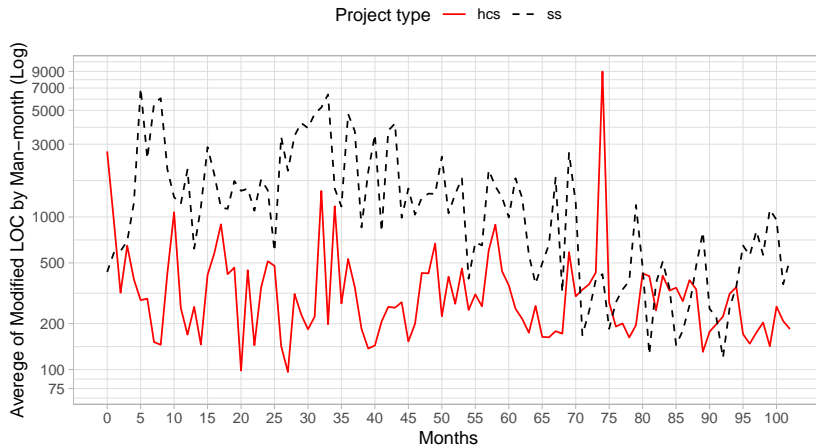


Figure 5.3: Man-Month productivity in test files per month.

and SS in terms of average number of modified files. However, our results show that the difference in effect size between HCS and SS is -0.181 (small), as can be seen in Table 5.3. We also observed that an SS contributor modify almost three times more test files (54) compared to a contributor in HCS (16).

We also investigated the workload distribution using an empirical distribution function (ECDF), as can be seen in Figure 5.5. Our results show a high concentration of workload since 20% of contributors are responsible for 91% and 97% of the work performed in HCS and SS, respectively.

20% of the contributors modify more than 90% of testing related files.

5.2.1.4 Assertions: Since assertions encapsulate testable logic of SUT and they are the focus of any test case, changes in assertions provide valuable information about the effort in developing and maintaining a test suite [118, 23]. We investigated the average number of assertions in the test files shown in Figure 5.6. The different types of project present an opposite behavior during evolution. While the proportion of assertions decreases in HCS, it increases in SS. The results show that the difference between the distributions is statistically significant in terms of number of assertions and the effect size is negligible ($|d| = 0.015$). Table 5.3 shows the result.

Additionally, we analyzed the density of assertions *i.e.*, the number of assertion statements by the number of lines of test code. This measurement has been used to evaluate the actual testing value that is delivered given a certain testing effort [22, 23]. Figure 5.7 shows the assertion density evolution, and we can observe that the density in HCS decrease over time. On the other hand, it increases in SS. Also, our results show that the distributions are statistically different in terms of assertions density, but the effect size is small ($|d| = 0.242$) as can be seen in Table 5.3.

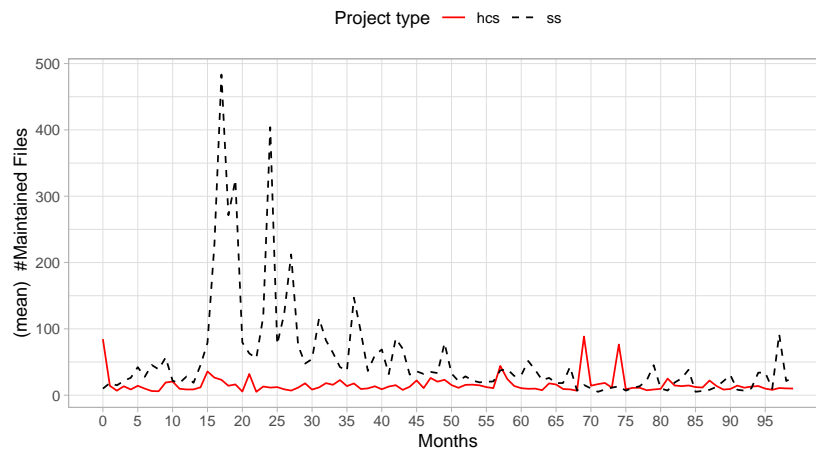


Figure 5.4: Modified test files per Contributors.

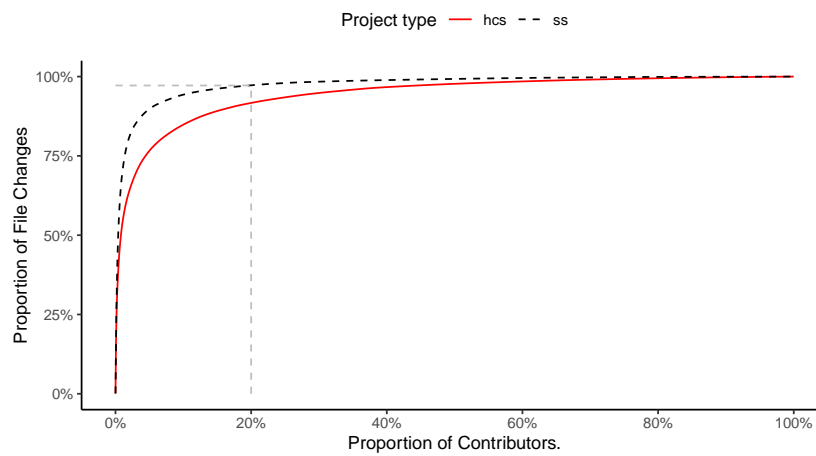


Figure 5.5: Distribution of Modified Files.

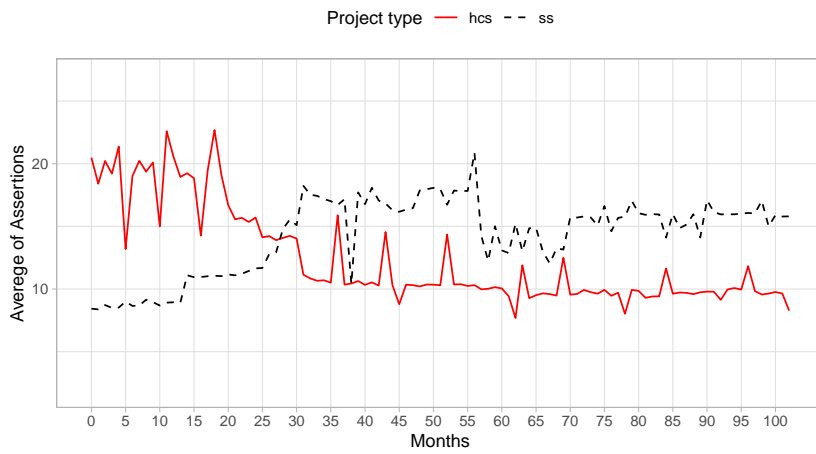


Figure 5.6: Average of assertions in Test Files.

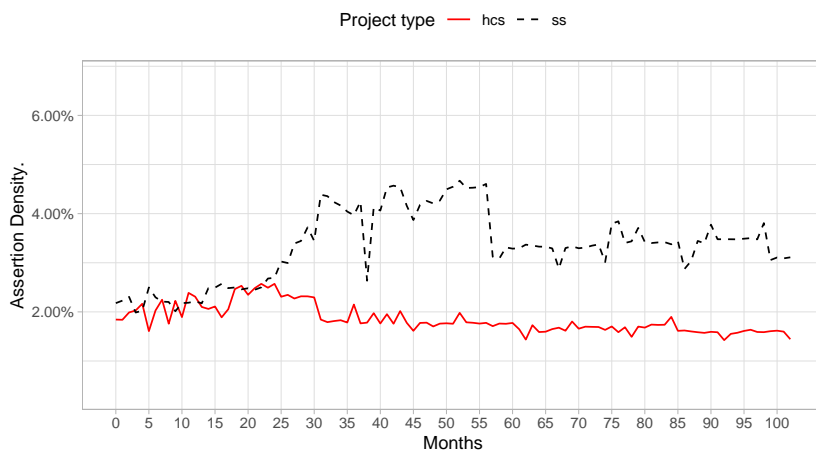


Figure 5.7: Assertions density in Test Files.

Over time, the average of number of assertions and density decreases in HCS and increases in SS.

5.2.2 RQ2 - How maintainable is the test suite and how that evolves?

Test code needs maintenance similar to production code [23]. Since our goal is to investigate how maintainable the tests in HCS are, we used existing maintainability measuring metrics proposed by Athanasiou et al. [23]. These include unit size, complexity, dependence, and duplication. These metrics are based on Software Improvement Group (SIG) quality model which is an operational implementation of the maintainability characteristic of the software quality model defined in the ISO/IEC 9126 [90].

5.2.2.1 Unit Size: Unit size is measured using the number of lines of code (LOC) in a unit [23]. The relation between LOC and maintainability is recognized both in the context of production code and test code [23]. According to Athanasiou et al. [23], as the lines of test code (*tLOC*) increases, the tests become harder to analyze.

Figure 5.8 shows the average unit size over time. We can observe that HCS present smooth growth with a small variation of unit size over time. On the other hand, SS presents severe fluctuations and downward spirals. Additionally, we analyzed the distributions of unit size in the test cases by system type, as shown in Table 5.4. Using the thresholds suggested by Athanasiou et al. [23], we can observe that 63% of HCS tests present a unit size less than 24 (lines of code), *i.e.* easy to maintain. In SS, only 19% of the tests low levels of unit size. On the other hand, 67% of SS presented very high values of unit size (unit size greater than 48), while 13% in HCS. This indicates that the maintenance of tests can become very difficult as the projects evolve, mainly in SS.

Some HCS tests have preprocessor directives, such as `#ifdefs` inside its structure, here called HCS-V. Whether we consider only these tests (HCS-V row in Table 5.4), the unit size values are similar to SS. For example, only 33% of the test files in HCS-V present low values, while 60% of test files present high or very high values. It indicates that variability inside the test code increases the unit size of tests. Therefore, tests with variability deserve special attention since the high percentage of tests with high and very high values of unit size can be a warning for the Obscure Test, and the Eager Test smells [101]. An obscure test is a test that has a lot of noise in it, noise that makes it hard to understand, and the consequences are that such test is harder to maintain and it does not serve as documentation [101]. An eager test attempts to test too much functionality. Additionally, our results show that the difference between the distributions is statistically significant in terms of unit size, and the effect size is large ($|d| = 0.598$), as can be seen in Table 5.3.

SS tests presented weaker test health than HCS tests when considered the unit size values of the tests.

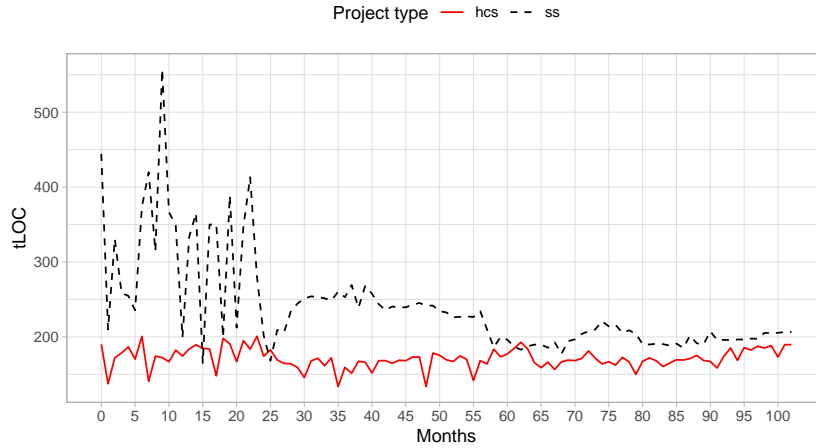


Figure 5.8: Unit size over time.

Table 5.4: Thresholds and Values for Unit Level Metrics.

	Unit Size				Unit Complexity				Unit Dependence			
	Low	Moderate	High	Very High	Low	Moderate	High	Very High	Low	Moderate	High	Very High
	24	31	48	>48	1	2	4	>4	3	4	6	>6
SS	19%	4%	10%	67%	44%	15%	17%	24%	57%	7%	8%	28%
HCS	63%	12%	12%	13%	0%	53%	26%	21%	79%	7%	4%	10%
HCS-V	33%	7%	12%	48%	0%	52%	23%	25%	31%	14%	23%	32%

5.2.2.2 Unit Complexity: The test quality model [23] proposes to use the *McCabe's cyclomatic complexity* [102] as measurement of unit complexity. However, for HCS, the variability implementation needs to be considered in the complexity measurement [89]. Thus, our results of unit complexity for HCS support the computation of *McCabe's cyclomatic complexity* metric, and the variability aspects on the code [89].

Figure 5.9 shows the average of the unit complexity of the tests in the different types of projects. Considering the threshold for test code originally suggested by Athanasiou et al. [23] (shown in Table 5.4), we observe that at the beginning of the tests lifetime, the test functions in HCS present very high values ($unit\ complexity > 4$), while SS present high values ($2 < unit\ complexity \leq 4$). However, as projects evolve, an opposite trend can be observed for the different types of projects. While HCS decreases the average unit complexity, SS increases the values. Nevertheless, HCS presents a high average unit complexity (HCS = 4.9). Since HCS presents a decreasing trend, we verified whether this reduction results from decreasing in the proportion of top functions, *i.e.*, functions with high or very high unit complexity. However, we did not find significant improvement in top functions from the first (21%) to the last month (19%). One must remember that the number of functions also increases with time. Thus, the reduced average value is just a result of having more functions with relatively lower complexity.

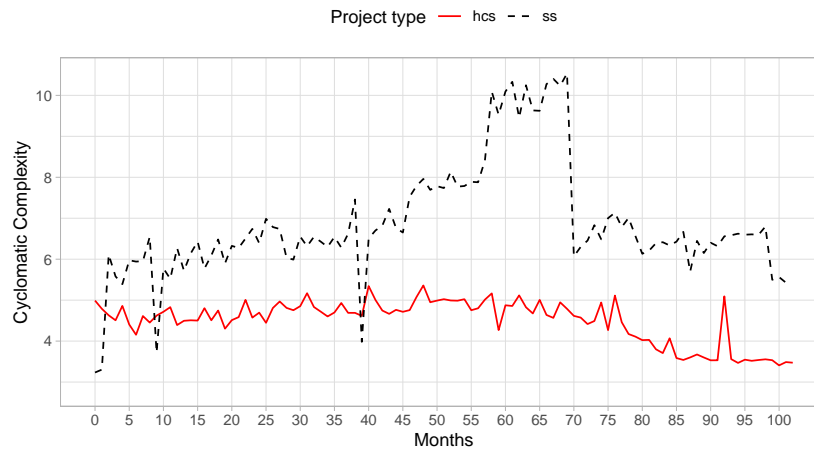


Figure 5.9: Average of Cyclomatic Complexity in test files.

We also analyzed the distributions in terms of unit complexity, as shown in Table 5.4. The results show that the difference between the distributions is statistically significant, and the effect size is medium ($|d| = 0.358$). Interestingly, HCS does not have low complexity values in the tests, but a substantial part (53%) presents moderate values. On the other hand, 44% of the tests in SS present low values and 15% moderate values. We can observe similar values for high and very high values for different types of projects. The high values for these categories raise an alert for the development teams since the complexity should be kept as low as possible to avoid writing tests for test code [23]. This is also underlined in the description of the Conditional Test Logic smell [101], which advocates keeping the number of possible paths as low as possible to keep tests correct and straightforward. The moderate unit complexity values are not so different for HCS when considering only test files with variability (HCS-V row). However, very high values increase in HCS-V. It indicates that the variability increases the complexity of the tests. The difference between the distributions is statistically significant (Table 5.3).

In general, the average unit complexity starts high in HCS and decreases as the projects evolve. An opposite trend was observed for SS.

Test cases with variability are more complex than HCS test without variability.

5.2.2.3 Unit Dependence: Module coupling measures the coupling between modules in the production code. In the context of test code, the modules' coupling is minimal [23]. Since dependency affects the changeability and the stability of the tests [23], it is a good indicator of required maintenance effort. We count the number of calls from a test code unit to production code units as a measurement of dependency [23].

Figure 5.10 shows the average of unit dependence per type of project during the evolution. In general, HCS increase slightly during evolution. Nevertheless, almost all

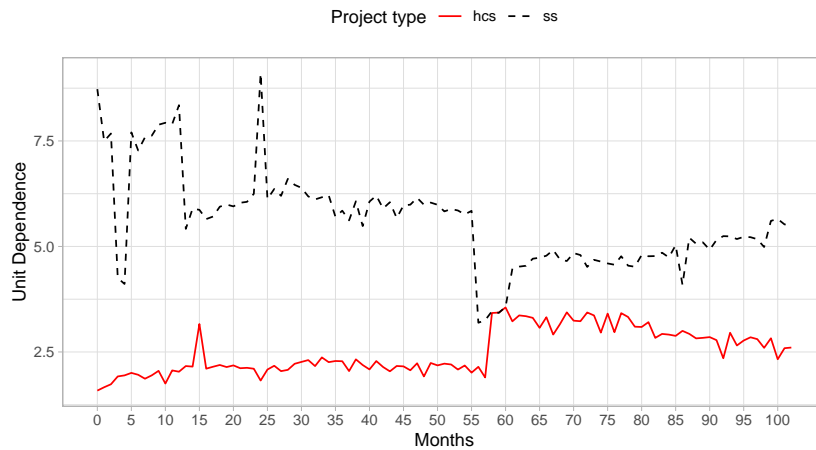


Figure 5.10: Average of Unit Dependence of the test files.

the time the average values are low ($unit\ dependence < 3$), considering the thresholds suggested by Athanasiou et al. [23] for test code. In SS, the evolution behavior of unit dependence is irregular but presents a decrease over time.

Additionally, we analyzed the distributions of unit dependency by system type, as shown in Table 5.4. Using this distribution, we can observe that 79% of the tests are classified as having low values of unit dependence ($unit\ dependency < 3$), and just 10% are presented very high values ($unit\ dependency > 6$) in HCS. On the other hand, SS present almost the triple of tests with very high values compared to HCS. However, if we consider only HCS tests with variability (HCS-V), the percent of test files that present high or very high values is the majority (55%). It indicates that variability in test files increases unit dependency. The high values of unit dependence in HCS-V can be explained since these files test different features and combinations, resulting in calls to other parts of the production code. The results give us a good indication that HCS-V tests are “complex to maintain”, *i.e.*, changes in the production code can propagate easily to the test code and cause tests to break (fragile test code smell [101]), increasing the test code maintenance effort. We also tested if the unit dependence in test files is statistically different between HCS and SS. Our results are positive, and the effect size is medium ($|d| = 0.414$).

Average unit dependence of HCS starts low and increases as the system evolves. The opposite is observed for SS.

The majority (79%) of HCS tests present low values of unit dependence. However, HCS tests with variability (HCS-V) show high values.

5.2.2.4 Duplication: Test code duplication occurs when copy-paste is used as a way to reuse test logic. It results in many copies of the same code, a fact that may significantly

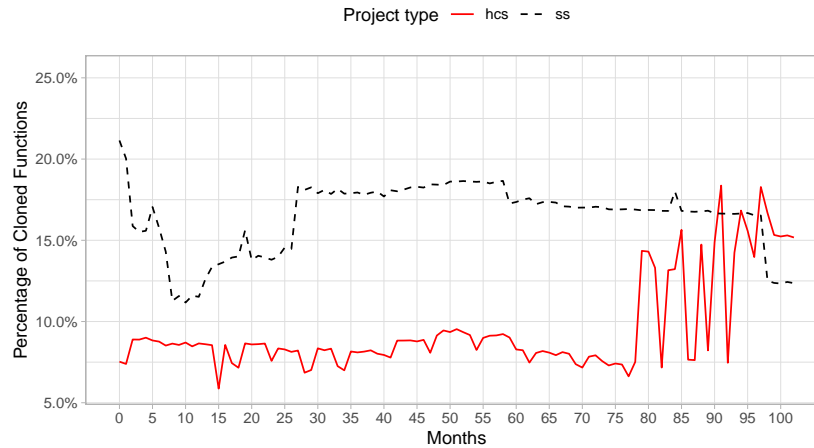


Figure 5.11: Percentage of Total Cloned Functions.

Table 5.5: Percentage cloned LOC by level of Similarity.

	% of Cloned LOC				Total
	S = 70%	S = 80%	S = 90%	S = 100%	
SS	4.8%	2.4%	0.9%	1.4%	9.5%
HCS	11.5%	2.8%	1.2%	1.3%	16.8%

increase the test maintenance cost [23]. This way, software clones can be considered harmful in software maintenance and evolution [122]. Moreover, test code duplication is identified as a test smell [101]. Duplication affects changeability, since it increases the effort that is required when changes need to be applied to all code clones. It also affects stability, since the existence of unmanaged code clones can lead to partially applying a change to the clones, thus introducing logical errors in the test code [23].

In this study, we identify a clone based on the percentage of all code that occurs more than once in at least 70% identical code blocks (functions/methods), ignoring white lines [105]. Figure 5.11 shows the percentage of cloned functions per type of project during evolution. We can observe that as the HCS evolve, the percentage of cloned test functions increases.

Also, we analyzed the distributions of duplication during evolution by different levels of Similarity (S), as can be seen in Table 5.5. For example, if the $S = 100\%$, it indicates the percent of only exact clones; if the $S = 90\%$, it indicates that two functions have at least 90% of the lines equal. The HCS present almost the double of functions cloned (16.8%) compared to single systems (9.8%). However, the numbers of functions cloned are very similarly for S different of 70%, in both types of systems. Additionally, we tested if duplication in test files is different between HCS and SS. Our results show that the difference is statistically significant, but the effect size is negligible ($|d| = 0.078$) Table 5.3.

We can observe that as the HCS evolve, the percentage of cloned test functions increases. An opposite trend can be observed for SS.

HCS has almost twice functions cloned (16.8%) compared to SS (9.8%).

5.3 DISCUSSION

In summary, the effort to evolve the test suite is high for both types of applications and becomes even more elevated for HCS over time. We identified that HCS has more contributors as it becomes mature. Thus, for better evolution planning, development teams have to be mindful of this need. Due to the high values of the test contributors' workload, development teams have to be watchful to this indicator, and whether necessary, should consider having more contributors only involved with test activities. This action can guarantee a better distribution of the tasks, and consequently, the workload reduction.

We observed that 96% of the contributors in both types of projects perform test and coding activities. Our results indicate that developers are writing the test cases themselves, and only a small portion of contributors (4%) are solely focusing on writing test cases. In general, this 4% of contributors have, on average less than five contributions. We posit that this small group comprises newcomers who are using this as a way of onboarding in the projects. However, further investigation is required to make any conclusive remarks and is an exciting research direction. Additionally, we observed a high concentration of work since 20% of contributors are responsible for 92% and 85% of the work performed in HCS and SS, respectively. Recent studies [123, 124] have shown that a high concentration of activities does not necessarily cause bottlenecks in development and communication or imply in the quality. However, test activities have different dynamics [23], and the research community needs to investigate whether the same occurs for tests.

We identified that HCS test contributors are less productive than SS. It can be explained by the presence of variability in code expressed in C-preprocessor directives or other aspects, such as feature interaction [108]. Our results have shown that these aspects could make the test activity in HCS more time-consuming. This way, in the analysis of HCS projects, researchers should focus on LOC and consider variability aspects in the evaluation. Moreover, our results show that variability increases the challenges in test development and maintenance, and contributors generally do not recognize (potentially harmful) test smells [125]. We believed that the creation of tools considering these specifics points in HCS could help the development teams control and improve upon their current work processes and productivity. Also, studies like [126, 127] to better understand productivity and work habits in HCS can be very useful.

The overall assessment of test code maintainability during the evolution is low for both types of projects. However, our results indicate that the variability in HCS tests makes maintainability still more difficult for these projects. For example, we observed that HCS tests with variability could increase the unit size considerably as the project evolves. Consequently, this kind of test is more prone to the occurrence of Obscure and Eager Test smells [101]. These smells can quickly increase the effort to maintain the test

suite. Some possible solutions for these smells in the context of SS are proposed in [101]. However, we identified that the opposite evolution behavior for unit size could complex the adoption of similar solutions for SS and HCS. This way, more investigation about the nature of high unit size in HCS needs to be performed to define suitable solutions.

We identified that the incidence of tests with high or very high unit complexity is considerable in SS (41%) and HCS (47%). These results show that as the projects evolve, the tests become susceptible to Conditional Test Logic smell [23, 101]. Moreover, we observed that the variability in tests increases the values of unit dependence in HCS. High unit dependence makes the tests more likely to occur fragile test smell [101] and generally, the presence of this smell increases the test code maintenance effort [101]. However, this smell detection process is not too straightforward for HCS written in *C*. In this sense, existing tools [128] should be extended to support the language and preprocessor directives used in HCS. In addition, new studies should be considered to investigate test smells and its impacts in the presence of variability. Other directions could be related to automated test cases refactoring. Some ideas in this direction are starting to be explored [129], however, we need more robust evaluations and tool support. Not enough, development teams need to deserve special attention to clones, mainly in the context of HCS since the tests present almost twice functions cloned (16.8%) compared to SS (9.8%). Code clone and detection tools are widely explored in singly system [130, 131, 132, 133]. However, even with some preliminary results [134], the HCS area lacks studies considering production and test code perspectives.

5.4 THREATS TO VALIDITY

We have taken care to ensure that our results are unbiased, and have tried to eliminate the effects of random noise, but it is possible that our mitigation strategies may not have been effective. In this section, we discuss the threats to validity for our study.

Construct Validity: The set of metrics used to evaluate test evolution was selected from the literature. However, we could have used other metrics. Thus, our evaluation is not complete, but we believe that the metrics that were used provide a fair assessment of test evolution. Although HCS and SS have different purposes, and we used the same metric to evaluate the test evolution in both domains, we mitigate this threat by taking into account specific characteristics of HCS in the evaluation.

External Validity: Although we have studied a limited number of projects, they represent a wide variety of different sizes and domains which provide sufficient data to address this topic.

Internal Validity: We used a set of scripts to crawler and extract information on the project's repositories. Nevertheless, these scripts can contain implementation faults. We control this threat by applying pair programming and extensively testing our implementations.

Reliability Validity: A simple heuristic was used to identify files with test code. Distinguishing test files from files of the system under test based upon naming conventions and folder localization might not be reliable. However, we performed a manual analysis on a sample of files that served as *(i)* a validation of the heuristic, and *(ii)* a sanity-check

for the reliability of our identification.

5.5 CHAPTER SUMMARY

The benefits of having a well-updated test suite are known [5, 12]. However, test evolution in configurable is still little explored. Using historical data from 36 open source projects, we empirically investigated test evolution in HCS and compare it with the test evolution in SS. In particular, we focus on two main aspects: effort and maintainability.

We conclude that the effort to develop the test suite is high and increases for both types of projects as they evolve. However, the variability present in HCS increases the challenges in the development of tests for these systems. We identified that the nature of the contributors' effort needs to be more explored. Additionally, we observed that the creation of specific recommendations and automated tools for testing activities in HCS are essential for effort reduction. Concerning maintainability, the overall assessment is low for both types of projects. However, the variability present in HCS tests makes the maintainability still more difficult for these systems. We identified that some differences between HCS and SS make it challenging to adopt a unique solution for both. Additionally, the development teams in HCS have to be watchful to tests with variability, since they are more prone to the occurrence of some test smells, such as: obscure test, eager test, fragile test, and test duplication smells.

We believe that our results and empirical observations contribute to the advance of the research concerning test evolution in HCS. Based on the identified causes that increase the effort and maintainability of the test suites, we provided some implications for development teams and researchers. Such implications can improve the current practices, tools, and make the software tests less expensive and time-consuming. Future research includes the investigation of the nature of the effort and maintenance of the test suite. We are planning to conduct an in-depth analysis of the contributors' work, including a survey and semi-structured interviews, to confirm or refute the findings identified.

The next chapter present an extended study to analyze the test suite evolution in another category of reusable systems. The main goal is to verify whether some observations are recurring and gather new data that support the findings.

TEST EVOLUTION IN A SOFTWARE ECOSYSTEM: THE EXTENDED STUDY ON NPM PACKAGES

In the previous Chapters 4 and 5, we provided systematic reports to get evidences about the test evolution in reusable systems. Also, we presented a set of empirical observations and implications for development teams and researchers. However, these observations maybe not sufficient to contemplate another category of reusable systems, such as ecosystems. This way, we performed an extended study to analyze the test suite evolution in an ecosystem to verify whether some observations are recurring and gather new data that support the findings. Although we are analyzing projects written in a different programming language and utilizing another implementation method to achieve the reusability, software development's social aspects are kept [135, 123]. Additionally, we enlarge the empirical studies performed in previous chapters by looking beyond the test evolution and exploring social software engineering.

Thus, we investigated the development and evolution of tests in popular provider packages in the `npm` ecosystem. We tried to provide empirical evidence about the existence of tests in such packages and to understand how tests change as the packages evolve. Additionally, provide a overview about the effort and maintainability of the test suite by analyze the contributors' workload to understand how the packages sustain its test development evolution. To conduct our study, we collected data from 1,691 popular packages in `npm`, the largest ecosystem supporting the `Javascript` programming language. `JavaScript` was the most commonly used programming language in 2018¹.

From the analysis, we identified and discussed ten observations, including the implications for community, practitioners, and tools developers. The remainder of the chapter is organized as follows: Section 6.1 explains the data collection procedures that we employed to process the studied `npm` data; Section 6.2 presents the results of our research questions. Section 6.3 discusses the implications of the observations; Section 6.4 discusses the threats to the validity of our study; and Section 6.5 concludes the chapter.

¹<https://insights.stackoverflow.com/survey/2018>

6.1 DATA COLLECTION

This section explains how we collected and processed the studied `npm` data. Three main steps were performed: package selection, repository mining, and file type classification. Figure 6.1 depicts an overview of our data collection approach.

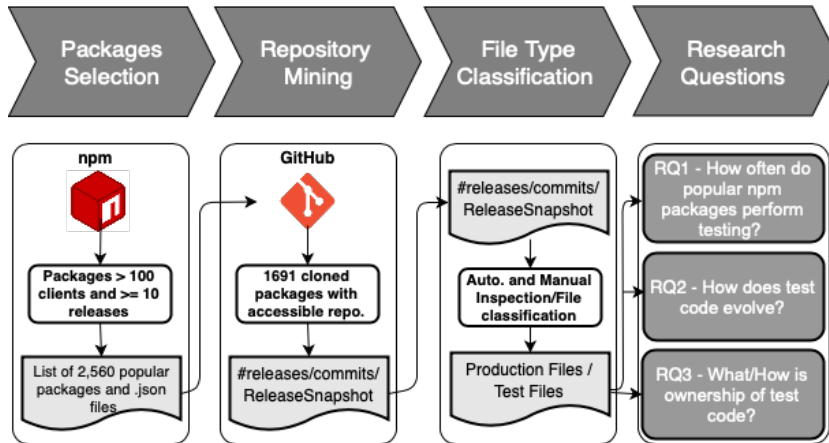


Figure 6.1: Overview of our data collection approach.

Package Selection: We crawled the registry of `npm` and filtered by packages that contain at least 100 clients (popular packages) and at least 10 releases. As a result we obtained the `package.json` metadata file of 2,560 popular packages. The metadata file of each package lists, among other pieces of information, all the published releases by a client package, the name of the used providers in each release, the associated versioning statements with the providers in each client release, and the timestamp of each release. Our data collection encompassed the period of December 20, 2010 to July 27, 2019.

Repository Mining: For each package we obtained its source code from GitHub. In some cases, the package publisher did not provide a GitHub link, in which case we could not obtain the source code with the commit history. In total, we mined 1,691 packages. The output of this process is thus the releases of each a list of commits and a list of associated files for each release (Release Snapshot).

File Type Classification: `npm` requires that developers provide a test script name with the submission of their packages (listed in the `package.json` file) [42]. However, developers can provide any script name under this field, and they do not need to specify a folder for this script. These situations make difficult to know if a package is tested and identify the associated test files. This way, we adopted two main steps in the file type classification: (i) to determine if a package has any associated test, we used a similar approach of other works [42, 136, 137] by looking for the term “test” (and its variants) in the file names and file paths; (ii) packages are formed by different types of files, such as production code, test code, notes, manuals, etc. Thus, we identified the most used script languages in `npm` popular packages, and only files of these types were considered as test

Table 6.1: File type classifications examples.

Production Code	Test code
any JavaScript (.js), TypeScript (.ts), and CoffeeScript (.coffee)	any JavaScript (.js), TypeScript (.ts), and CoffeeScript (.coffee)
All the files except test files	file that has “test” (and its variants) in the file names and file paths

or production code files: JavaScript (.js), TypeScript (.ts), and CoffeeScript (.coffee). Table 6.1 provides example classifications for some types of files.

Tools Selection: In the sections 6.2.2 and 6.2.3, we evaluated some aspects of evolution related to Lines of Code (*LOC*) (*i.e.*, lines of code related to files that contain system functionality) and test Lines of Code (*tLOC*) (*i.e.*, lines of code related to files that code related to tests). To measure the *LOC* and *tLOC* we used *scitools Understand*². This tool offers metrics reports, and its reverse engineering feature provides a complete structure of the code dependence. In addition, this tool provided other information, such as complexity, duplication, and etc. These information can be used in future investigations.

6.2 RESULTS

6.2.1 RQ1: How often do packages perform testing?

Motivation: Popular provider packages are used by a large number of clients. Therefore, a defect in these providers can cause a failure in a large number of client packages. Moreover, prior studies show that one of the clients motivation to use a popular provider is the perception of quality [73, 42]. Since testing plays an important role in the quality assurance process, in this RQ we study the extent to which tests are performed by popular provider packages.

Approach:

- We calculate the proportion of popular provider packages that have at least one release containing a test file. In this evaluation we use the file type classification described in section 6.1. Also, we determine whether popular provider packages with tests have more clients than the ones without test. To compare the distributions, we test the null hypothesis that both distributions do not differ from each other using the Wilcoxon Rank Sum test ($\alpha = 0.05$) [119], and assess the magnitude of the difference with the Cliff’s Delta (d) estimator [120]. To classify the effect size, we used the following thresholds [121]: *negligible* for $|d| \leq 0.147$, *small* for $0.147 < |d| \leq 0.33$, *medium* for $0.33 < |d| \leq 0.474$, and *large* otherwise.
- We calculate the number of releases taken to a package start to have test.

²<https://scitools.com/>

- To evaluate the most used test frameworks by the popular provider packages, we adopted some steps: 1) fetch the list of frameworks from grey literature³, 2) search the used dependencies by popular provider packages to verify whether they use any of the listed frameworks, 3) calculate the proportion of framework usage. In the last step, we take into account that one provider package can use more than one test framework at once. This way, the sum of percentages related to used frameworks can be higher than 100%.
- Additionally, we compare the difference in the number of updates and downgrades of popular provider packages that have and do not have tests. To account for the discrepancy in number of client package of each provider package, we measure the downgrades and updates using normalized values. First, we count the number of client package of each provider package. Then, we divide the number of downgrades/updates by the total number of client packages. We used Wilcoxon Rank Sum test, and Cliff's Delta to compare the distributions.

Findings: Observation 1) *We observed that 14.5% of the packages do not have any test script.* This observation is showed in the Figure 6.2. We identified that from all different clients packages, 32.5% use providers packages that do not have test. Additionally, we identified that the median number of client packages for provider packages that have test is higher (272) than the provider packages that not perform tests (210). However, the difference between the distributions is not statistically significant and the effect size negligible ($|d| = 0.101$).

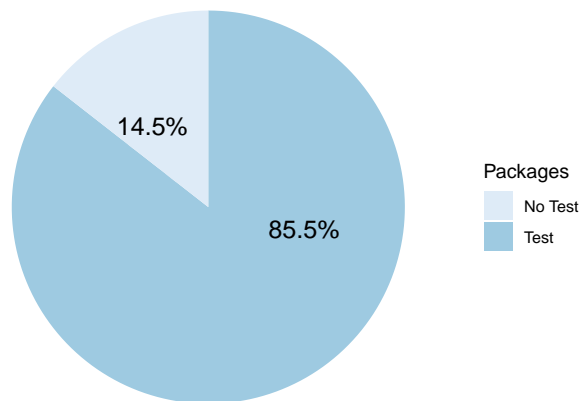


Figure 6.2: Proportion of popular packages that perform or not tests.

Observation 2) *Half of the packages introduce test in the first or second release.* This observation is depicted in Figure 6.3. Interestingly, 14.5% packages take more than 10 releases to introduce tests.

³<https://medium.com/welldone-software/an-overview-of-javascript-testing-in-2018-f68950900bc3>

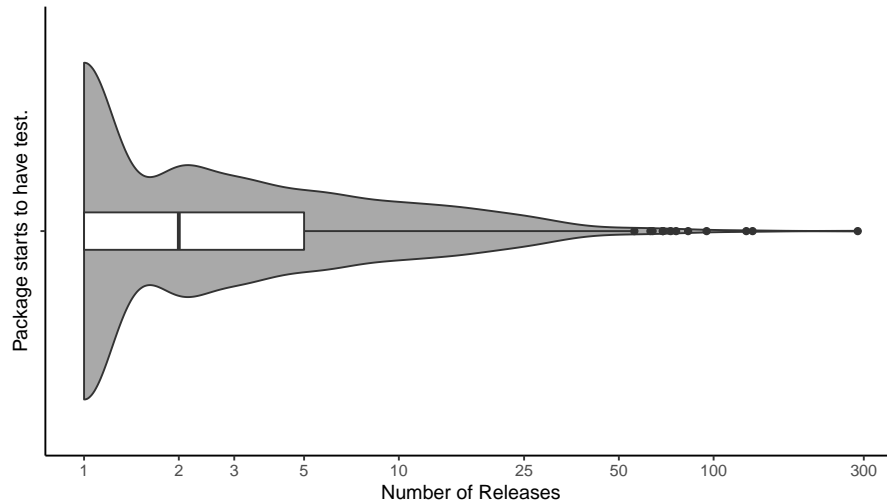


Figure 6.3: The number of releases taken for a package to start having tests.

Observation 3) *29.1% of the packages that perform test use some test frameworks being mocha the most commonly used.* Figure 6.4 shows the distribution of test frameworks adopted by popular provider packages. We identified that the Mocha is the leading test framework used in about 89% of the popular packages that contains test files.

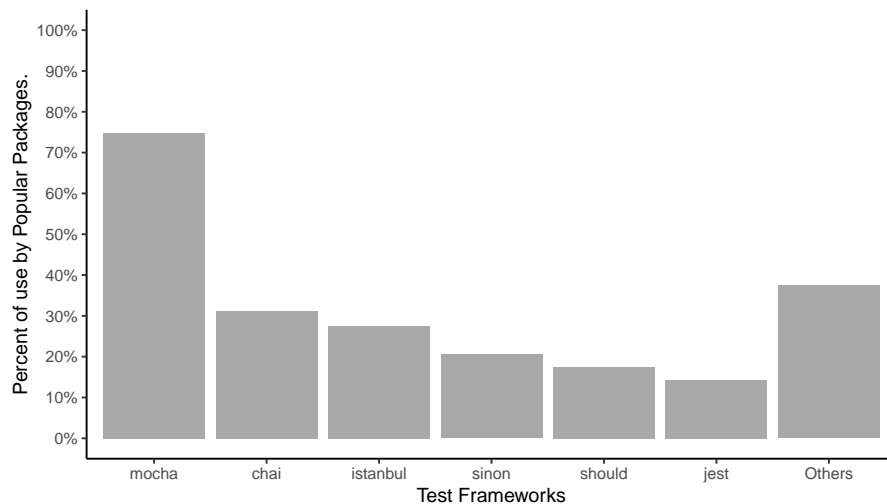


Figure 6.4: Test framework adoption distribution by popular packages.

Observation 4) *Our results suggest that there is no difference of downgrades by client packages between releases of providers packages that have or not tests.* Figure 6.5 shows that client packages perform less downgrades in releases of provider packages that have test. However, the difference between the distributions is

not statistically significant and the effect size negligible ($|d| = 0.040$). Additionally, we identified that upgrades of provider packages by client packages are not associated with the presence of tests in providers packages. In the same way of downgrades, the difference between the distributions is not statistically significant and the effect size negligible ($|d| = 0.034$).

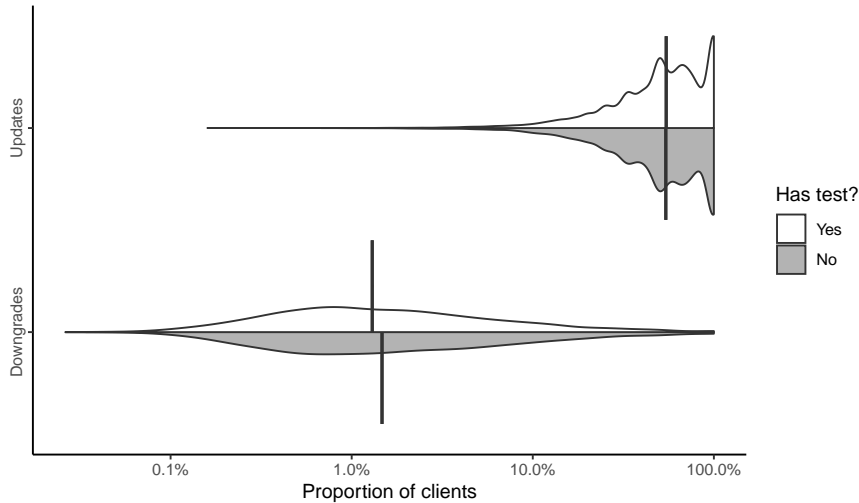


Figure 6.5: Proportion of Downgrades and Updates for releases with and without test.

Summary of RQ1. We identified that 14.5% of the popular packages do not have any test script after ten releases. In particular:

- In general packages introduces tests in the first or second release.
- The adoption of test frameworks is low (29.1%) by the packages that perform test activities.
- There is no statistical difference of downgrades by client packages between releases of providers packages that have or not tests.

6.2.2 RQ2: How does test code evolve?

Motivation: Packages evolve, among other reasons, due to bug fixes, new features, and refactoring. These changes can be followed by changes in the test code [138]. The evaluation of the performed changes, as well as the stability of such tests, can reveal valuable information about the test evolution [5, 23]. In particular, understanding how test changes over time is important for development teams to allocate personnel and resources to test tasks effectively and to reduce the test overhead in regular development tasks, such as fixing defects and adding new tests. Therefore, in this RQ we study the test evolution of popular provider packages.

Approach: We aim to verify how the test suite size evolves. Therefore, we only considered in our analysis packages that have tests. Typically, studies about software evolution require a time-frame to be adopted in the comparison. Researchers have previously chosen to use releases within a given unit of time [114, 115], individual commits, or discrete-time units (such as years, months, weeks, or days) [116, 19]. Since the release schedules of the packages are different, a time-based slicing wouldn't work. This way, we divide the lifetime of each project in five different time-frames, namely: “Beginning-release”, “One-quarter-release”, “Middle-release”, “Three-quarter-release”, and “Latest-release”. For example, as can be seen in Figure 6.6, in a package with 11 releases, the first release is the Begin, the fourth release is the One-quarter, the sixth release is the Middle-release, and etc. Each time-frame of a package is compared with the same time-frame of the other packages. To evaluate the test suite size distributions in each package release time-frame, we used the Scott-Knott Effect Size Difference (ESD) [139] to cluster such distributions into statistically distinct groups with a non-negligible difference.

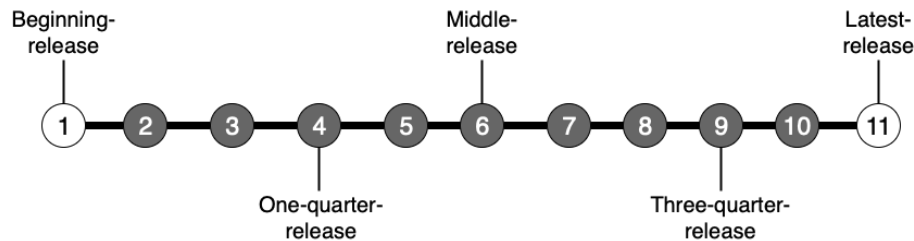


Figure 6.6: Example of the division of a packages' time-frame.

Findings: Observation 5) *After tests being introduced, the number of test files and $tLOC$ do not present any significant evolution.* As shown in Figure 6.7, the test suite increase from 33% in the Beginning-release to 42% in the Latest-release. The same behavior is observed for the $tLOC$, as can be seen in Figure 6.8. The $tLOC$ goes from 24% in the Beginning-release to 48% in the Latest-release. However, applying the EDS over the distributions shown in Figures 6.7 and 6.8 show two distinct groups. Only the Beginning-release present a statistically significant difference compared with the other releases, both for the number of test files and $tLOC$.

Since the statistical difference between the releases time-frame is not significant (except for the Beginning-release), we verified what is the size of changes after a package introducing tests. To verify the difference, we plot the standard deviation of the difference, as can be seen in Figure 6.9. The results indicate that, in general, after a package introducing tests there is no significant change in terms of the number of test files and their size.

Observation 6) *A median of 80% of the test files are modified in each time-frame of the evolution, and each one of these changes modified 27 lines on average per test file.* This observation is depicted in Figure 6.10 and 6.11. As we expected the high value of changes occurs mainly in packages that contain few test files. We do not find distinct groups by applying the ESD in the distributions of packages

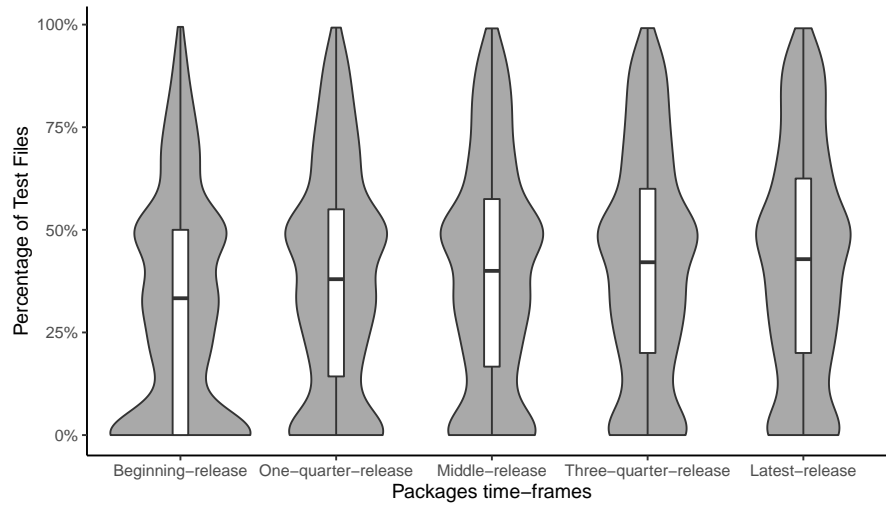


Figure 6.7: Percentage of test files.

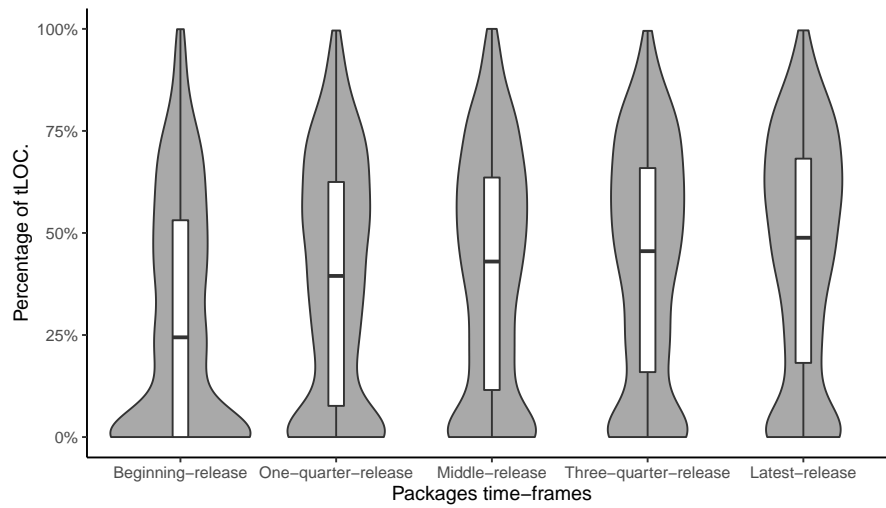


Figure 6.8: Percentage of *tLOC*.

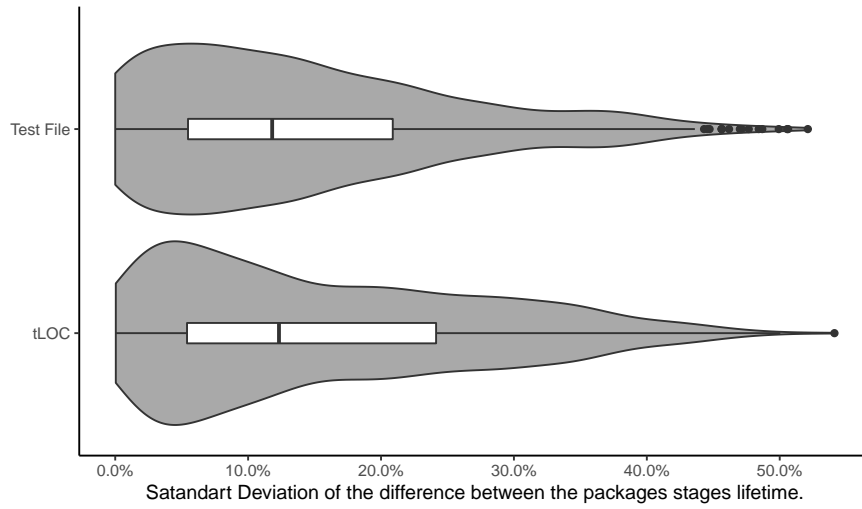


Figure 6.9: Percentage of *tLOC*.

time-frames.

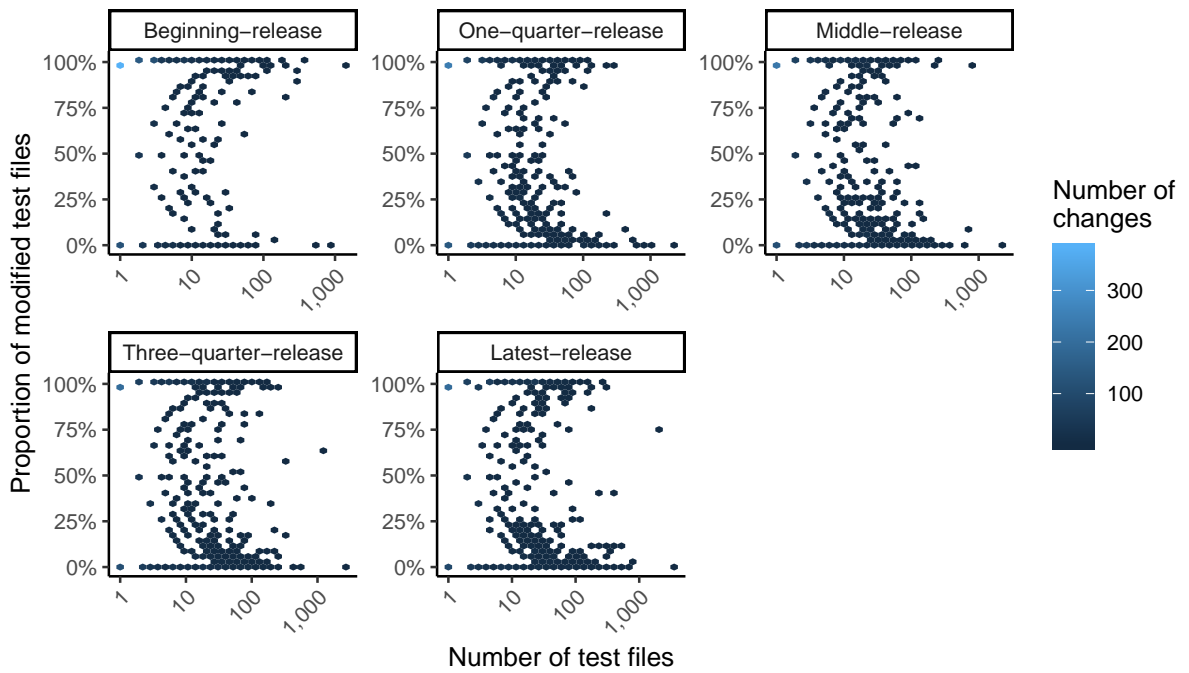


Figure 6.10: Proportion of modified test files during each time-frame.

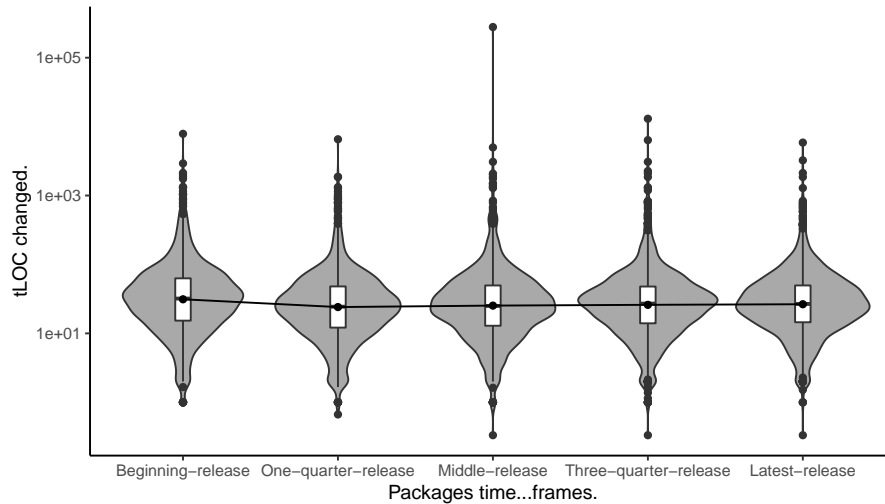


Figure 6.11: Average of tLOC changed per test file.

Summary of RQ2. Although the packages show an increase in the median values of test files changes, we do not identify any significant evolution of the tests after the One-Quarter of the package timeline. In particular:

- The test files accounts for up to 33% of all project files in the begin, and 42% in the end.
- The percentage of *tLOC* increases two times in the median from begin (24%) to the end (48%) of the packages time-life.
- 80% of the test files are changed in each stage of the evolution.

6.2.3 RQ3: How is the ownership of test code?

Motivation: The development of open source software ecosystems requires forms of balancing the workload between groups of participants [25]. Typically, the groups are divided into a small core group that does most of the work and coordinates a much larger group of peripheral participants. As an ecosystem grows, it is not clear whether the arrangements that previously made the ecosystem work will continue to be relevant or whether new arrangements will be needed. The analysis of ownership concentration can provide the community with an overview of the “sustainability” of popular packages, and specifically, how sustainable are the tests in these packages. This analysis is important since failure in a popular package can cause a massive instability on the ecosystem. An anecdotal example is the incident with a package called `left-pad`, which was removed from the ecosystem and caused failure to some of the largest Internet sites, *e.g.*, Facebook, Netflix, and Airbnb [42]. The massive failure occurred because the `left-pad` packages was used by a popular package called `Babel`. While the real reason for the `left-pad` incident

was that npm allowed authors to unpublish packages, it could be avoided by the presence of tests in the popular package by checking the presence of the dependence. This way, the analysis of ownership in the test activities can help to understand how popular packages sustain its test development evolution and the ecosystems as a whole.

Approach:

- To study the number of test contributors in the packages during the evolution, we compare it against that of the source code. The comparison is necessary since the test suite is a much smaller component than the source code (6.2.1). This way, we normalize the values by measuring the proportion of contributors in the test and source files during the evolution. First, we count the number of contributors in test and source files in each package time-frame. Then, we divide each count by the total number of contributors in test files or the total number of contributors in source files that existed in the time-frame. We repeat this process for each time period.
- To investigate the workload of the contributors involved with test activities over time, we adopted two steps. First, we count the number of test lines of code (tLOC) created and modified in each time-frame. Second, we divide each count by the total number of unique contributors in test files in the time-frame.
- To evaluate the percentage of One-time Contributors (OTC) *i.e.*, contributors that contributed only one time, we count the number of contributors with just one commit in the package and divided by the total number of contributors in the package.

Findings: Observation 7) *The proportion of contributors involved in test activities is stable during the evolution. However, the contributors' workload is increasing.* Figure 6.12 shows the proportion of test contributors evolution. We can observe that the median proportion of contributors in test files is very regular, and our results do not show a statistical difference between the distinct groups by applying the ESD in the distributions of packages time-frames. One must remember that the percentage of test files and *tLOC* are increasing (section 6.2.2). Since we do not identify a relevant increase in the percent of test contributors, we check the contributors' workload during the evolution, as can be seen in Figure 6.13. The median of *tLOC* by unique contributors is increasing. However, in the same way as contributors, our results do not show a statistical difference between the distinct groups by applying the ESD in the distributions of packages lifetime.

Observation 8) *20% of the test contributors (core group) are responsible for 97% of work.* This observation is depicted in Figure 6.14. Our results indicate a high concentration of test activities in the popular packages. Part of this high concentration can be attributed to 7.8% of the packages that have only one contributor.

Observation 9) *In the median 52% of the contributors are One-time Contributors.* This observation is depicted in Figure 6.15. Interestingly, in 2% of the packages, all the contributors are OTC. Investigating this scenario, we identified that, for such packages, tests were introduced in a late version, in the median at eighth release. This

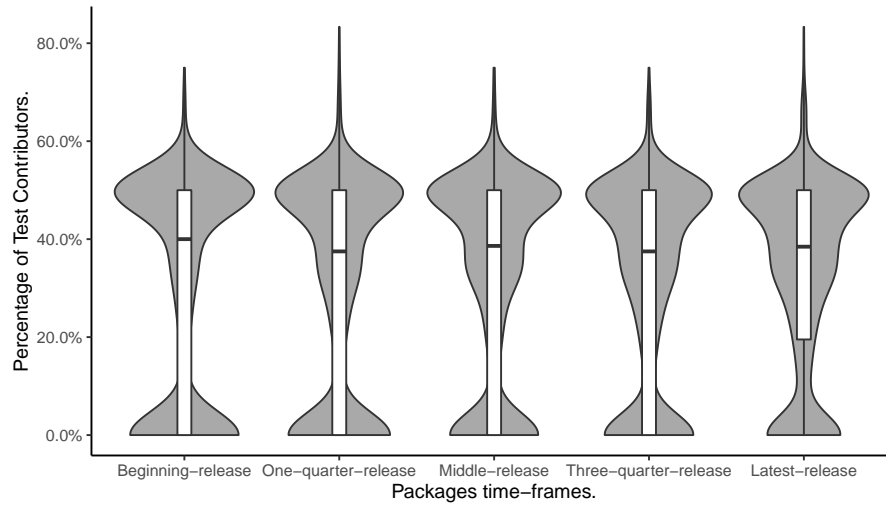


Figure 6.12: Percentage of test contributors.

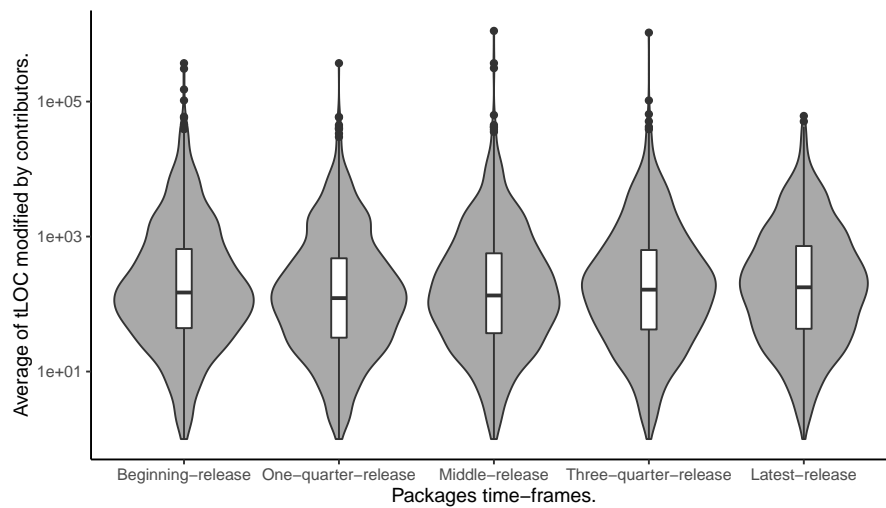


Figure 6.13: Average of modified *tLOC* by contributors.

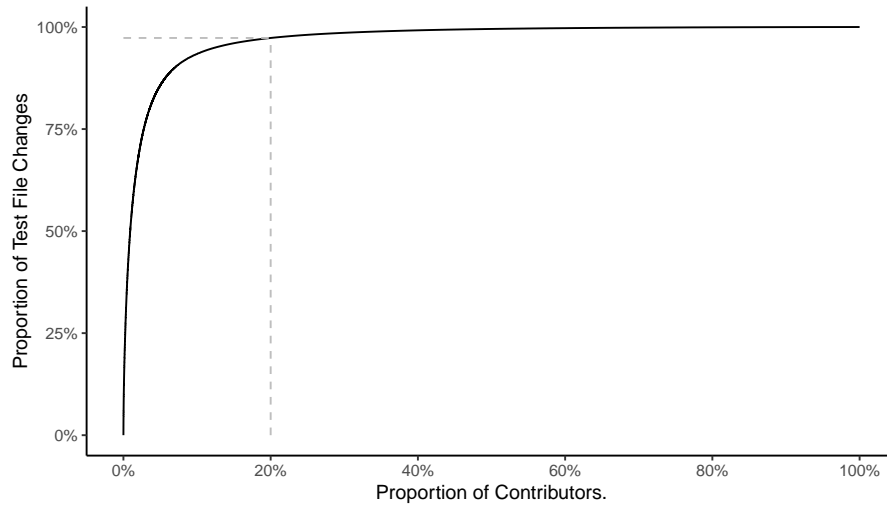


Figure 6.14: Distribution of modified test files.

media is higher than for all packages that were 2 (section 6.2.1). Additionally, we identified that test contributors in packages that have only OTC are composed of less than five developers.

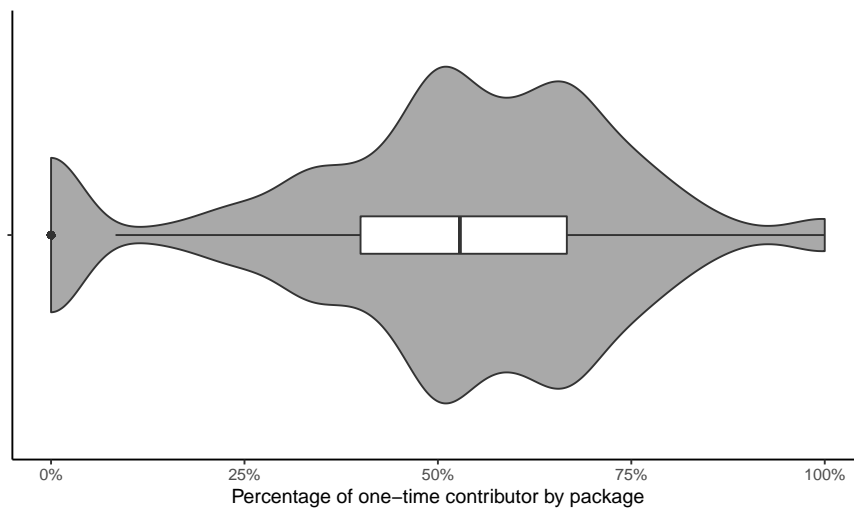


Figure 6.15: One-time contributor (OTC).

Observation 10) *37% of the test files do not have changes after its creation.* Figure 6.16 shows the percentage of test contributors that perform changes in a test file. Interestingly, 28% of the test files have a single contributor, i.e., only one contributor changes this file along the evolution.

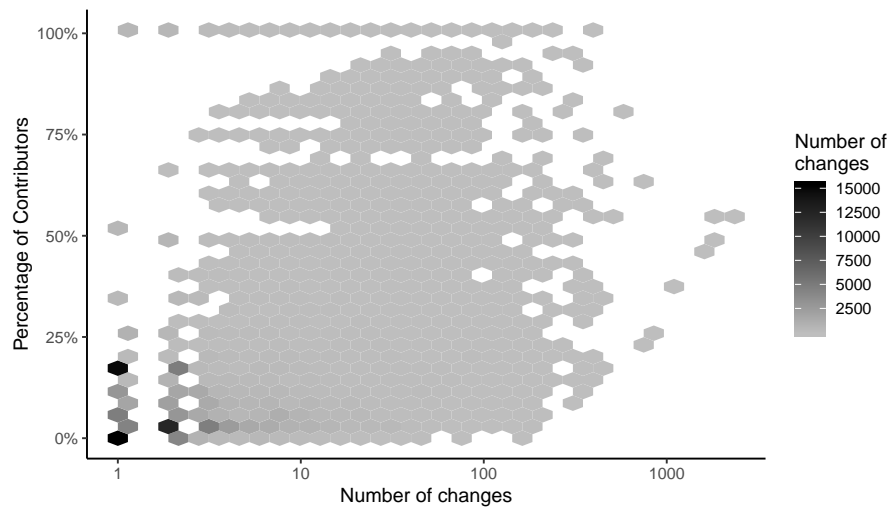


Figure 6.16: Percent of contributors that perform changes in the test files.

Summary of RQ3. The ownership of test code is highly concentrated in small group of contributors, and a considerable amount of the test files have "owners". In particular:

- The proportion of contributors involved in test activities is stable during the evolution.
- The high concentration of the test activities has been contributed to increases the workload in the core group.
- One-time Contributors are the majority in the packages, and this factor contributes to the concentration of work.
- 28% of the test files have a single contributor along the evolution.

6.3 DISCUSSION

In this section we present the implications to the observations in the results.

Implication 1) *Popular provider packages developer should increase their testing code (section 6.2.1).* One of the most cited reasons for using these packages is that they provide a well-implemented code with a credible assurance of quality. This way, the client packages' assumption (perceived quality) is wrong (at least from the point of view of testing), and the association between popularity and perceived quality doesn't hold. We conjecture that clients are either unaware of this fact or they simply do not care. This situation is concerning since the lack of tests becomes the packages more error-prone, and a failure in a popular package can cause massive instability in the ecosystem Abdalkareem:2017. However, more investigations are necessary to evaluate how much the lack of tests affects the quality of the packages.

Implication 2) *Package developers should plan for the introduction of tests since the start of a project.* Once the quality of popular provider packages is an important factor for the stability of the software ecosystem, the presence of test, in the beginning, can avoid that a defective popular provider propagates errors to a large number of client packages. Additionally, client packages need to avoid the adoption of a provider package just based on the number of users of this package. A good strategy for client packages is to analyze if the provider package performs tests and check the history of reported problems and downgrades. In short and medium-term, the adoption of these steps can avoid unexpected issues and downgrades.

Implication 3) *Provider packages should consider the usage of test frameworks.* Test frameworks are an essential part of any successful automated testing process and can improve the test quality. One concerning situation is that a significant part (89%) of the packages that use the test framework adopted the `mocha`. So one failure or breaking change in this framework can result in a catastrophic situation that can invalidate the test suite of the packages. This way, the adoption of more than one test framework at the same time can mitigate this problem.

Implication 4) *In spite of client packages perform less downgrades in releases of provider packages that have test, our results suggest that there is no statistical difference (section 6.2.1).* Even so, development teams of client packages need to be watchful to the use of providers packages that do not perform test, and the teams have to avoid the adoption of "fresh releases" of these providers packages. Additionally, almost all packages that have tests were downgraded, indicating that in-house tests are as good as no-test-at-all to capture in-field failures. However, more investigations are necessary for this direction, since some downgrades can be preventive.

Implication 5) *We identified a slight growth of the test suite with stable periods over time (section 6.2.2).* This indicates good testing health of the packages and suggests that the testing-process is under control Zaidman:2008. However, further investigation is required to evaluate if the coverage offer by the test suite is sufficient and to make any conclusive remarks about the quality of the tests.

Implication 6) *Development teams must be very diligent to ensure that the test suite does not become defect-prone as a result of high values of changes in the test files. (section 6.2.2).* The high value of changes suggests that test files are continually evolving Nagappan:2005, McIntosh:2011. However, the high value is concerning since prior research on source code has shown that frequently changing in modules make these modules contain more defects than slowly changing ones Nagappan:2005. This way, the monitoring of changes by the development teams is essential since a defect-ridden test suite can no guarantee the quality of the software.

Implication 7) *The proportion of contributors involved in test activities is stable during the evolution (section 6.2.3).* As a result, we observed that the contributors' workload is increasing. It is a concerning situation since defects in popular packages generally need a quick fix, and this scenario can make a quick fix harder. Development teams have to be watchful to this situation, and whether necessary, should consider having more contributors only involved with test activities. This action can

guarantee a better distribution of the tasks, and consequently, the workload reduction.

Implication 8) *Packages contributors should constantly evaluate the balance of workload between them.* The high concentration of work (section 6.2.3) can be an alert related to the quality of code. Although the development of open source software ecosystems requires forms of balancing the workload between groups of participants Zhou:2017. Typically, the groups are divided into a small core group that does most of the work and coordinates a much larger group of peripheral participants. As an ecosystem grows, it is not clear whether the arrangements that previously made the ecosystem work will continue to be relevant or whether new arrangements will be needed. Recent studies Amritanshu:2017, Majumder2019 have shown that a high concentration of activities not necessarily cause bottlenecks in development and communication or imply in the quality. However, test activities have different dynamic Athanasiou:2014, and the research community needs to investigate whether the same occurs for tests. Additionally,

Implication 9) *Package contributors need to be watchful to the alterations made by One-time Contributors.* The high value of One-time Contributors indicates that a great part of the test contributors is not familiar with the test code. According to [140] contributors often struggle with code that they are not familiar with. However, test and source code tend to co-evolve Zaidman:2008, Athanasiou:2014, i.e., changes to the source code often require changes to the test files, and vice versa, a novice developer may easily introduce a source code change, unaware that a test change may be necessary. If the test is not changed when a change is required, it can cause tests to break (fragile test code smell Meszaros:2006), and increases the test code maintenance effort Athanasiou:2014. For this reason, it is essential the support of tool that maps the dependence between the production code and test code to assist new test contributors in the identification of code changes that require changes in the test suite.

6.4 THREATS TO VALIDITY

Internal Validity: We used a set of scripts to crawler and extract information on the project’s repositories. Nevertheless, these scripts can contain implementation faults. We control this threat by extensively testing our implementations and each script was reviewed by at least two authors.

External Validity: We have collected data exclusively from `npm`. Although `npm` is representative in size, and we had analyzed a wide variety of packages with different sizes and domains, our results may not generalize to other ecosystems. However, our approach can be replicated in other ecosystems.

Reliability Validity: We use a simple heuristic to identify files with test code. Distinguishing test files from production files based upon naming conventions and folder localization might not be reliable. Even though this technique is widely accepted in the literature [42, 136, 137], to confirm whether our technique is correct, we performed a manual analysis on a sample of files that served as (i) a validation of the heuristic, and (ii) a sanity-check for the reliability of our identification.

6.5 CHAPTER SUMMARY

Although we had analyzed the test suite evolution in another category of reusable systems(ecosystem), we observed that some evolution pattern are recurring, such as: a slight growth of the test suite with stable periods over time, a high concentration of work, the proportion of contributors involved in test activities. Additionally, we gathered new data that support the findings in previous steps. In summary, our empirical results led us to conclude that: *Not all popular packages perform test activities, and the client packages' "belief" (perceived quality) is wrong (at least from the point of view of testing), and the association between popularity and perceived quality doesn't hold at all.*

The next chapter presents a survey with test contributors to investigate the test evolution from the development point of view concerning the results obtained in this chapter and the previous ones.

SURVEY WITH TEST CONTRIBUTORS TO UNDERSTAND THE TEST EVOLUTION

In previous studies (Chapters 4, 5, and 6) presented in this thesis we have investigated the test evolution in reusable systems. However, our preliminary results reinforce the need of additional systematic reports to provide more evidence to understand the test evolution considering the software reuse. Thus, this chapter presents a survey with the test contributors, to investigate the test evolution from the point of view of development. The survey combines ideas from [141] with good practices from [142], providing an expert perspective in relation to results obtained by previous studies (Chapters 4, 5, and 6).

The remainder of the chapter is organized as follows: Section 7.1 describes the related work in the area. The methodology used is described in Section 7.2. Section 7.3 describes the results of the study and shows the contributors' perception. Section 7.4 presents the discussion of the results and the summary of findings; Section 7.5 presents threats to validity, and finally Section 7.6 presents a summary of the chapter.

7.1 RELATED WORK

In Chapter 3, we identified that the test evolution issues and concerns in configurable systems context have not been surveyed, compared, or documented in a systematic way. This way, we performed a set of empirical studies in reusable systems to obtain a better understanding regarding to the test evolution and providing new findings for further research (Chapters 4, 5, and 6).

Studies focusing on surveys in software reuse context artifacts have been presented in the past ([143, 144, 145]). Jha et al. [143] surveyed how software reuse is adopted in the HCS context to provide the necessary support for engineering applications. Additionally, they identified some issues related to the implementation of HCS. However, the study does not present any rigorous survey elicitation process. In [144], the author designed a survey to provide information regarding the vital process business area and specific HCS aspects. Although these studies are surveys in HCS, they do not consider the test aspects in their evaluation.

Daka et al. [145] surveyed 225 developers to understand unit testing practices such as motivation of developers, their usage of automation tools, and their challenges. Kochhar et al. [146] performed surveys on open-source and industrial practitioners to understand the test automation culture of mobile app developers.

In this context, our research is the first to analyze the test evolution aspects in reusable systems. Moreover, it was performed to explore findings from previous studies (Chapters 4, 5, and 6), achieving more reliable and generalized results. The study definition and reporting were also structured based on [141] according to well-defined guidelines.

7.2 METHODOLOGY

The method used in this research encompasses a survey with some aspects of expert opinion. According to [141], surveys are probably the most commonly used research method. Although surveys are a popular method of data collection, they must be used under the appropriate conditions [147]. In [147], the author states that a survey is not just the instrument (questionnaire or checklist) for gathering information. It is a comprehensive research method for collecting information to describe, compare or explain knowledge, attitudes, and behaviors.

This way, we systematically structured the survey using the five steps proposed by [141], as can be seen in Figure 7.1, and that are described as follow:

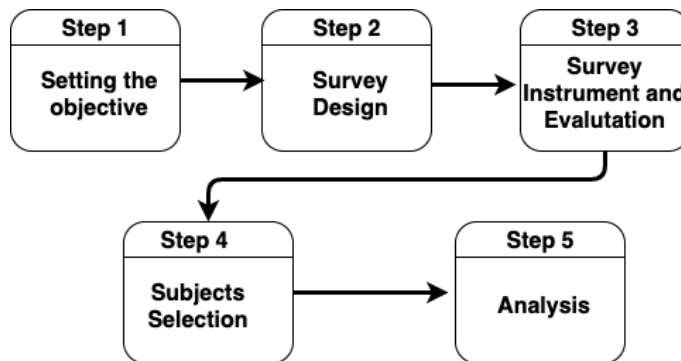


Figure 7.1: Survey Steps.

Objective: This survey aims to understand the test evolution and the developers' thoughts to improve and advise researchers and practitioners. The survey considers the perspectives in relation to the results of our preliminary studies (Chapters 4, 5, and 6) for this propose, we surveyed test developers who contributed to the projects analyzed in these studies.

Design: According to Fink [147], different surveys design serves different objectives, and two purposes can be distinguished: experimental and descriptive. The experimental design is characterized by arranging to compare two or more groups, at least one experimental. Descriptive designs produce information on groups and phenomena that already exist. Regarding how the survey can be applied, it can be classified [141] as: self-administered

questionnaires, telephone surveys, and one-to-one interviews. In this study, we adopted a descriptive self-administered questionnaire since the subjects were not co-located.

Survey instrument and evaluation: The survey was developed following the steps designed by [141, 147] that includes: search the relevant literature, construct, evaluate, and document the instrument. To achieve the survey goal, questions involve issues related to: contributors' role in the projects, effort to develop the test suite, maintainability aspects, and ownership.

According to [147], a question can take two primary forms. When they require respondents to use their own words, they are called open. When the answers or responses are pre-selected for the respondent, the question is termed closed. Both types of questions have advantages and limitations. In [141], the authors advise that in self-administered questionnaires, questions are mainly closed, and open questions could be included to complement the answers. Our survey questionnaire contains 15 mixed questions *i.e.*, open and closed questions, to be concluded in less than 10 minutes¹. To mitigate the possibility of introducing misunderstood or misleading questions, we performed a pilot with three test developers and incorporated their minor suggestions to improve the survey.

Subjects: We selected all the test contributors of the projects analyzed in the Chapters 4 and 5. The survey was emailed to the 879 selected developers. However, some of the emails were returned for various reasons (*e.g.*, the email account does not exist anymore, etc.), we could only reach 254 developers. We received 31 responses to our survey, which translates to a response rate of 12.2%. Our survey response rate is in line with, and even higher, than the typical 5% response rate reported in questionnaire-based software engineering surveys [148].

Analysis: According to [142], when aggregation methods are being used, they range from easy-to-use methods, such as simple averaging of several inputs, to more complex techniques, such as, the classical method [149] and Bayesian aggregation [150]. Studies in forecasting demonstrate that aggregation of opinions is necessary since they indicate that aggregated opinions, even if only a simple average, is consistently better than individual opinions. Arithmetic and geometric averages, for instance, are two aggregation methods used in practice. Although these methods presuppose that all participants are equally weighted, which is a strong assumption, it does not limit the use of the aggregation method by the researchers [142].

We adopted the arithmetic average aggregation method based on the judgment justified by our analysis of contributors' responses, in which no significant biases were introduced in the study. Moreover, all contributors were equally weighted during the aggregation since no significant difference was observed in contributors' credibility and importance.

7.3 RESULTS

In this section, the collected data analysis is presented, discussing each issue in detail besides some correlation.

¹Survey questionnaire: <http://doi.org/10.5281/zenodo.4157137>

7.3.1 General Information

Initially, we asked for general information about the contributors surveyed in the study. We identified that of the 31 respondents, 84% are professional software engineers paid by companies (Figure 7.2). Interestingly, 52% of the test contributors have an advanced educational qualification (Master's, Ph.D., M.D.), and only 4% have less than high school, as shown in Figure 7.3. Regarding development experience, 42.3% of the respondents have more than ten years of experience, 24.6% have between 6-10 years, and 19.2% have 1-5 years of experience (Figure 7.4). The fact that most of the respondents are experienced developers gives us confidence in our survey responses.

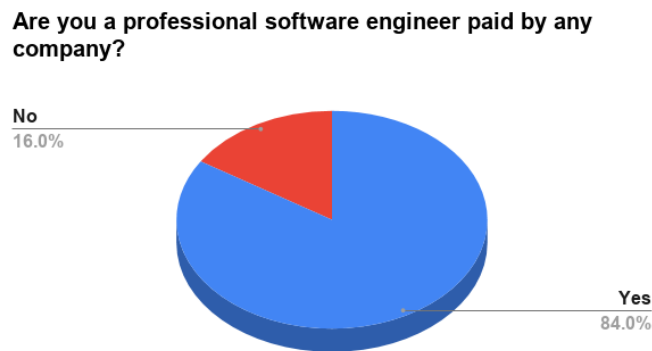


Figure 7.2: Contributors Information.

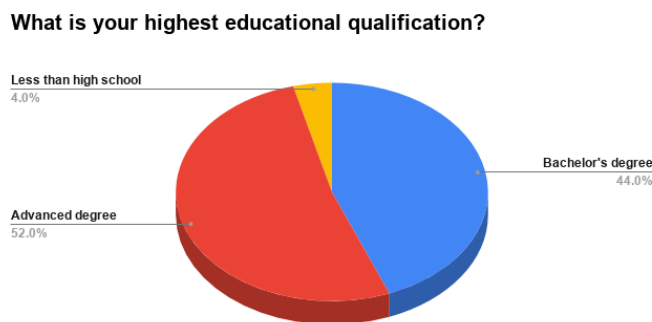


Figure 7.3: Educational Qualification.

7.3.2 Effort Characteristics

Test Contributors at the beginning of a project. In the previous studies (Chapters 4, 5, and 6) we identified that considerable effort on tests is spent in the early stages of

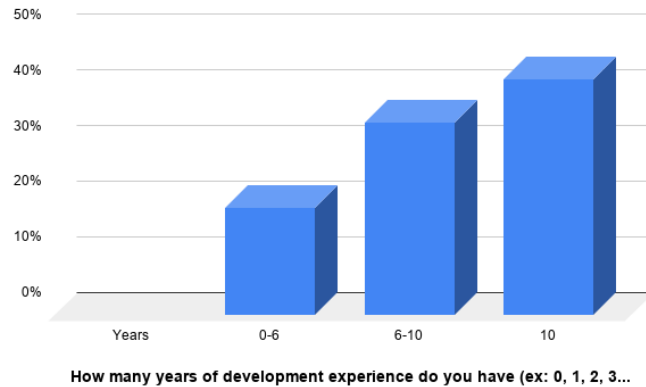


Figure 7.4: Contributors Experience.

project due to initial implementation. This observation is reinforced by the contributors since 50% of them agree or strongly agree (Figure 7.5) with the necessity of more test contributors at the beginning of a project. Additionally, they were asked about the necessity of three times more test contributors at the beginning of a SS project than HCS (Chapter 5). According to them, this vast difference does not make sense since the tests are essential in both systems. However, one contributor reported (C5²): “This difference can be explained since tests are more straightforward in SS, and this way, more people are doing the test in this kind of project.”

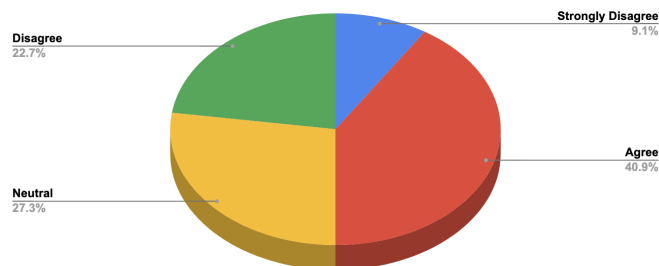


Figure 7.5: The necessity of more contributors at the beginning of a project.

Contributors’ workload. High workload was a concerning situation identified in the previous studies (Chapters 4, 5, and 6). This way, we asked the test contributors if they agree that the workload is high in HCS. 77% of the test contributors agree that the number of lines changed in HCS is high. One survey contributor highlight that (C9): “while HCS can reduce some code duplication, the code in theses systems also adds more code/build system and run-time parameterization. As a result, one change in the code could result in more alterations for different parts of the code”. Additionally, we asked

²Here we are referring to the contributors of the survey as C5 for contributor number five.

if the test contributors believe that the high workload negatively impacts the quality of tests (Figure 7.6). 37.5% of the test contributors strongly agree, and 31.3% agree that the high workload negatively impacts the quality of the tests. The contributor **C19** highlights that “defects in HCS generally need a quick fix, and high workload can make a quick fix harder and contributes to the introduction of more errors”.

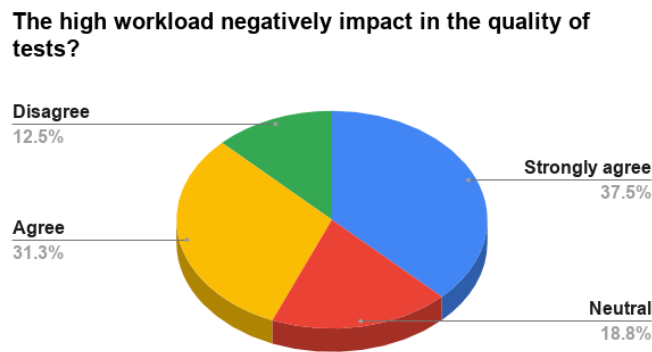


Figure 7.6: The negative impact of high workload.

Contributors’ for only test activities. The test contributors were also asked about the necessity of more contributors only involved with test activities since we had identified high values of workload in test activities (Chapters 4, 5, and 6). 31.3% of the test contributors agree, and 43.5% strongly agree with this statement, as shown in Figure 7.7. On the one hand, the contributors argue that as HCS are more complex, changes generally result in more alterations in other parts. This way, if the project has contributors only involved with the test activities, they could contribute to more precise tests. On the other hand, the Contributor **C8** argued that “*restricting interested contributors to only working on tests could be a detriment since they are not familiar with the production code. As a result, the tests can become more error-prone*”.

The concentration of test work. In Chapter 5, we identified a high concentration of work since 20% of contributors are responsible for 92% in HCS. Recent studies [123, 124] have shown that a high concentration of activities does not necessarily cause bottlenecks in development and communication or imply in the quality. However, test activities have different dynamics [23], and more investigation is necessary. This way, we asked the test contributors if the high concentration of activities impacts development and communication. 43.8% of the contributors surveyed strongly agree, and 37.5% agree that a high concentration of test activities negatively impacts development and communication. One contributor (**C13**) highlighted that: “*the high workload values are a consequence of just a few test contributors have a great understanding of the project required to contribute effectively to the tests*”. Additionally, the contributor (**C21**) highlighted that: “*High concentration is bad any development activity in the long term, and It can be an alert related to the quality of code.*”

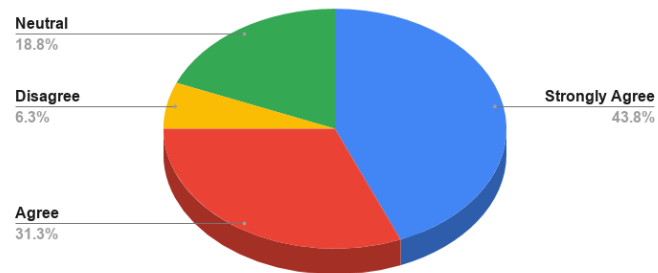
Necessity of more contributors only involved with test activities.

Figure 7.7: Contributors only involved in test activities.

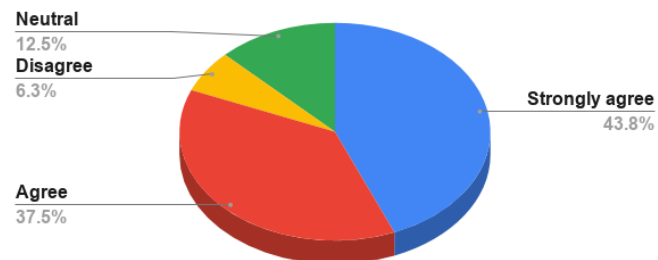
The high concentration of activities as consequences of bottlenecks in development and communication

Figure 7.8: Concentration of work.

7.3.3 Maintainability sub-characteristics

Code Complexity. In Chapter 4, we identified that the test code presents very high values of unit complexity, and they are not easy to maintain. Additionally, we observed in Chapter 5 that test cases with variability are more complex than the HCS test without variability. Therefore, to obtain the test contributors' perception, we asked them if they agree that the HCS test's code complexity is higher than SS. 76% of the test contributors agree (24%) or strongly agree (52%) with this statement (Figure 7.9). They argue that HCS implementation (like `#ifdef` annotations) generally introduces more instructions making the code more complex. Additionally, the Contributor **C29** highlights that *“the code complexity (the control flow of the code) is not the main problem related to complexity. He stated that the code relationship (the feature scattering) is much more problematic. As a result, it increases the test code complexity considerably.”*

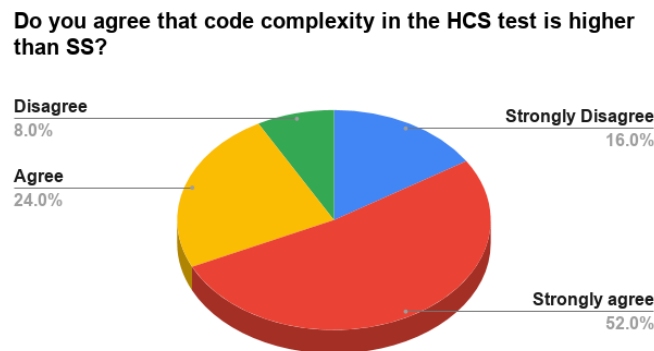


Figure 7.9: Code Complexity.

Code Dependence. In Chapter 4, the average of the unit dependence presented low values, and the results were encouraging. However, some test programs showed values totally out of the scale. This way, we also investigated the values of dependence in a set of HCS and compared it to SS (Chapter 5). We observed that the variability in tests increases the values of unit dependence in HCS. Therefore, we asked if the dependence of tests in HCS is higher than SS due to variability. The test contributors (72%) strong/strongly agree that the variability raises dependence (Figure 7.10). One contributor (**C11**) reported that the high values of unit dependence in HCS could be explained since *“these files test different features and combinations of them, resulting in calls to different parts of the production code”*. The perception of test contributors indicates that variability present in HCS tests and the dependence makes them “complex to maintain” *i.e.*, changes in the production code can propagate quickly to the test code and cause tests to break. This increases the chances of test smells [101] and raises the test code maintenance effort.

Code Clones. We identified a high percent of cloned functions (Chapter 4) in configurable systems. Since the clones are a great indicator of test smell [101], we extended the

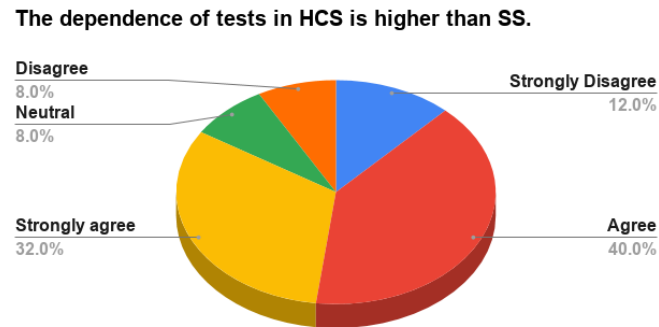


Figure 7.10: Code Dependence.

investigations in a set of configurable systems and compared them to SS. We observed that SS has the triple of functions cloned compared to configurable systems. When asked the test contributors about the perception of clones in configurable systems and SS, 65% of the test contributors responded that this difference was already expected. They argued that configurable systems code is structured in a reusable way, reducing code clones, while SS generally does not apply any systematic reuse. On the other hand, the Contributor (C29) highlights that *"although the presence of clones in test code in configurable systems is less than SS, the complexity present in configurable systems make the clones present in these systems more dangerous."* We also highlight that the tool support for clone detection in single systems is widely explored [130, 131, 132, 133], while configurable systems presenting only some preliminary results [134, 151, 152].

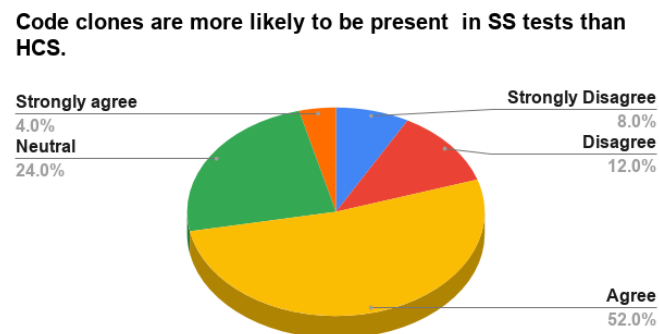


Figure 7.11: Code Clone.

7.3.4 Main Challenges

We also asked the test contributors about the main challenges of testing configurable systems based on their experience. This way, we obtained important aspects of the test development that the questionnaire does not cover. The main concern highlighted by the contributors is the variability inside the tests. 57% of the contributors cited the variability as the main problem in the test development in configurable systems, and the most cited reason among the variability aspects is the feature interaction. The contributor (C22) states that *“the greatest difficulty is featuring interactions, as there is a disparity between them, and among other interaction problems.”* Additionally, the contributor (C04) highlights that the code complexity is improved as a result of the interaction of features. In fact, determining the influence of feature interactions on a system’s behavior has been a challenging subject for decades [108]. Although tool support has been presented [108], it does not cover the test code.

Another aspect highlighted by the test contributors was the lack of existence of smell detection tools. The contributor (C09) states that *“based on your experience some errors in the tests are recurrent, and the use of tools can avoid them”*. In fact, as we had identified in Chapters, 4 and 5, some test smells are more prone to occurrence in reusable systems, and the test smell detection is not straightforward in these systems.

Test coverage was also cited as a current problem. The contributor (C18) explains that code coverage and fault detection effectiveness are not direct in configurable systems due to the variability inside the test artifacts *i.e.* the coverage of a test suite is not correlated with its size since the tests may not identify the variability aspects. Additionally, the contributor (C25) highlights that it is challenging *“identify whether the tests are covering the correct code interaction.”* Although different coverage tool has been proposed [5, 63], these tools do not consider the interaction among features in the analysis.

7.4 DISCUSSION

The study results reinforced the previous results (Chapters 4, 5, and 6). Some discrepancy exists which can be explained by analyzing some particularities. Concerning the aspects of the test evolution survey, we do not have much to compare since this is the first one with such a strong focus on test evolution in reusable systems.

As a result of the initial effort of implementation in configurable systems, the test contributors reinforced the necessity of more test contributors at this time of a project. Open-source teams need to be aware of this demand and be prepared to avoid high workloads since it is a concerning situation identified in the previous studies. According to the test contributors, the high workload is recurring in the test development of reusable systems, impacting the test quality negatively. The test contributors agreed that the adoption of contributors only involved with the test activities could contribute to the workload reduction and more precise tests. However, restricting the interest of contributors to only working on tests could be detrimental since they are not familiar with the production code. Although the high concentration of activities does not necessarily cause bottlenecks in development and communication or implied in the quality of developing

production code, it does not hold for the test contributors' perception.

According to the contributors, the complexity in configurable systems is high, and it is increased by the implementation that generally introduces more instructions making the code more complex. Moreover, the code relationships (the feature scattering) and feature interactions introduce more complexity to the code beyond structural complexity. Concerning test code dependence, the contributors believed that the joint between dependence and variability increases maintainability considerably. As a result, the contributors highlight the increases in the chances of test smells occurrences and the maintenance effort. Although clones in configurable systems are not higher than SS, the lack of tool support for clone detection in configurable systems and the variability in the code make the clones more dangerous in these systems.

Interestingly, the test development's main challenge based on the contributors' perception is the variability of the test code. They highlight the feature interaction, and the lack of a tool that could help them deal with it makes test evolution more time-consuming and increases the cost. Additionally, the lack of tool support for test smell detection and code coverage, considering the variability aspects, is also highlighted by the test contributors as a problem. In fact, tool support is essential in any development activities to avoid the introduction of errors, reduce costs, and improve quality.

7.5 THREATS TO VALIDITY

There are some threats to the validity of our study, which are briefly described and discussed:

Internal Validity: The most obvious threat to internal validity is the sample size, which is small when applying statistical inference-based analysis methods. The small sample size is a consequence of our labor-intensive data collection method based on expert opinion and the belief that it was more important to get high-quality, in-depth information rather than lower quality. Additionally, most of the respondents are experienced developers, giving us confidence in our survey responses.

Construct Validity: This question addresses whether the measures had some stability across methodologies, i.e., that the data reflect accurate scores of artifacts chosen. We sought to improve the construct validity with some steps. Thus, we conducted pilot tests, which led to the questionnaire's modifications to improve both the content and construct validity. Moreover, during the pilot, we asked the respondents to improve the questionnaire, e.g., removing some items and adding new ones.

External Validity: As in any empirical study, our study also has threats to external validity. Our results may not hold for all test developers. An additional sample of test developers may result in some different results. To mitigate the risk due to this sampling, we surveyed HCS test contributors in various projects and domains.

7.6 CHAPTER SUMMARY

In this study, we investigated the contributors' opinions to provide the required information to understand the test evolution from different perspectives, such as effort, main-

tainability, variability, and general aspects.

The arithmetic average aggregation method was used to gather the results obtained from the judgment justified by our analysis of test contributors' responses. The findings found in this study show to be consistent when comparing with previous ones.

Finally, this study also contributed by defining and improving the way to perform and report surveys. A guideline for experimental studies was also used during the study definition, which was very important to conduct and report the results, enabling further extensions and the study replication.

The next chapter presents the research synthesis and evaluation of the Multi-method approach performed in this thesis. Moreover, the strengths and weaknesses of the multi-method approach and lessons learned are discussed.

PART IV

CONCLUSIONS

RESEARCH SYNTHESIS AND EVALUATION OF THE MULTI-METHOD APPROACH

In Section 1.4, we introduced the multi-method approach used in this work based on empirical software engineering ideas. The process aims to advance the test evolution body of knowledge in reusable systems, evaluating characteristics that indicate the effort to develop and maintain the test suite, identifying reusable aspects that can affect the tests by analyzing the available evidence, the good practices, and lessons learned. The multi-method approach was initiated with an informal literature review (Chapter 3), which provided an overview of the existing research related to this thesis's subjects and presented related work to our investigation. The goal was to map out the test evolution field in reusable systems, synthesize available evidence to suggest important implications for practice, and identify research trends, open issues, and areas for improvements. Then, we started the second phase, which was composed of four empirical studies. First, we performed a case study (Chapter 4) to analyze the test evolution of a large configurable system. Next, we conducted a comparative analysis (Chapter 5) to evaluate the test evolution in 18 open-source projects from various sizes and domains in configurable systems and their similarities and differences to 18 SS projects. The comparison allowed us to identify similarities that can help us adopt techniques from SS to configurable systems and vice versa. The differences could help us design new strategies addressing the discrepancies. Third, we performed an extended study (Chapter 6) to analyze the test suite evolution in another category of reusable systems to verify whether some observations are recurring and gather new data that support the findings. Finally, we surveyed (Chapter 7) test contributors to investigate the test evolution from the development point of view and improve the findings in previous stages. The survey study extracted evidence from experts and built a perception of reality compared to the evidence extracted in earlier studies. In general, the evidence extracted during previous studies was confirmed, and identified new evidence.

In this chapter, we present the synthesis of the findings identified and validated in this Thesis. As we conducted a triangulation of data and collected quantitative and qualitative

results, a research synthesis was applied to summarize, integrate, combine, and compare our findings from different studies to understand test evolution in configurable systems.

This Chapter is organized as follows. Section 8.1 discusses the justification for adopting the multi-method research approach. Section 8.2 presents the research synthesis methods applied in this Thesis. Section 8.3 and 8.4 summarize, respectively, the findings of this Thesis and the applied multi-method approach. Section 8.5 discusses the results and analysis of this research based on the research questions. Section 8.6 discusses the benefits and drawbacks of applying this multi-method approach. Finally, Section 8.7 summarizes the Chapter.

8.1 JUSTIFICATION FOR THE MULTI-METHOD APPROACH

To achieve more reliable and generalizable results, an empirical study requires confirmatory power, reached by (i) external replication and (ii) investigation of the same phenomenon through a different empirical study [27]. The most effective way to achieve the second goal is through an approach that involves different methods. This way, the findings found during the first study are refined and investigated through the next studies. The results from each study can, therefore, turn out to confirm one another.

Comparing with the single-method approach (and to a large extent with the same method replication approach), an evolutionary application of the multi-method approach offers the following benefits:

- **Robust conclusion:** Results demonstrated by the multi-method phases are likely to be robust [27]. Using a combination of research methods applied to different population samples, conclusions are less likely to be affected by poor design, sample bias, misapplication of statistical tests, etc. Critically, the complementary nature of the research methods helps address individual techniques' relative weaknesses of individual techniques [27].
- **Increased understanding:** The multi-method approach can result in a deeper appreciation of the important factors affecting the phenomenon under investigation [27]. Different research methods, collecting different data types, are more likely to provide broad coverage of the problem space and alternative insights. As the program becomes more focused, a better understand should be developed. Just as important, differing results provide an opportunity for improved understanding through detailed investigation of the causes of inconsistency [27].

Consequently, results emerging from a multi-method approach are more impressive than those from a single empirical study. In turn, the software engineering community is likely to have more confidence in their reliability.

8.2 RESEARCH SYNTHESIS

The application of a multi-method approach to investigate the test evolution in configurable systems had provided an interesting and important set of empirical results,

which has shown consistency across the phases of the study (Chapters 4, 5, 6, and 7). However, credibly integrating multiple studies is not simple [153]. A classic approach to understanding what several studies say about some phenomenon is to conduct triangulation of information, a qualitatively existing results summary, and manually synthesize them [153]. The drawback of this approach is that it lacks a precise method for research synthesis.

Research synthesis is a collective term for a family of methods used to summarize, integrate, combine, and compare the findings of different studies on a specific topic or research question [154]. One of these methods of research synthesis is called Narrative Synthesis [155]. The Narrative Synthesis is a general framework of selected narrative descriptions and primary evidence that includes commentary and interpretation combined with specific tools and techniques that increase transparency, and trustworthiness [155]. A characteristic of narrative synthesis is adopting a narrative (as opposed to statistical) summary of primary studies' findings. According to [154], narrative synthesis can be applied to quantitative and/or qualitative research reviews.

The multi-method approach applied in this thesis synthesizes the evidence combining the classic and narrative synthesis techniques. This way, we investigated each source of information to derive each finding in the study and present the results in a narrative form. For each finding, we analyzed which source of evidence/empirical method contributes to identifying that finding, reporting it was the original data source or a confirming source. We also analyzed whether any data sources suggested contradicting evidence.

8.3 SUMMARY OF THE FINDINGS

In this section, all the findings are summarized. Table 8.1 was developed based on [156] and shows a summary of the empirical studies. The findings were classified based on different categories. Firstly, they were classified according to the research method, which exposes it: case study (**CS**) (Chapter 4), comparative study (**CP**) (Chapter 5), extend study (**ES**) (Chapter 6) or survey (**SR**) (Chapter 7). As the research methods were performed sequentially, the first time in which evidence was found was marked with “**S**” (Source of information). If confirmed the evidence in another method, a “**C**” was used to categorize as a confirming evidence. If the finding was identified and confirmed in the same method, in this case, the finding was marked as “**SC**” (source and confirmation). A finding can also be marked with more than one “**S**”, which means that there are alternatives research methods responsible for the finding or research methods used to produce the findings. A “**CE**” indicates that a finding of the research method type is contradictory to the source of the finding.

F1 - The effort to develop a test suite is high. By analyzing the effort metrics and tabulating their trends over time (in Chapters 4, 5 and 6), we observed a continuous growth in size, and functional content. Although the number of contributors increases, the effort of development does not decrease over time. The perception of test contributors (Chapter 7) reinforced this finding since small changes could result in other alterations in the related code.

F2 - More effort is spent in the early stages of a project. We identified considerable

Table 8.1: Summary of Findings.

	F#	Brief Description	CS	CP	ES	SR
Effort	F1	The effort to develop a test suite is high.	S	C	C	C
	F2	More effort is spent in the early stages of a project.	S	C	C	C
	F3	The effort increases as the project evolve.	S	C	CE	C
	F4	Contributors' workload is high during evolution.	S	C	C	
	F5	More contributors have to be only involved with test activities		S	C	
	F6	Not all projects include tests at the beginning.		S	SC	
	F7	Although there is a perception of quality in reusable systems from a testing perspective, we identified not all projects perform tests.			S	
	F8	Lack of Test framework considering the reusable aspects.	S		C	
Maintenance	F9	The test code maintainability is low over the evolution.	S	C		C
	F10	Some test smells are more prone to occurrence in reusable systems.	S	C		C
	F11	Lack of tool support to the specific context of reusable systems.	S	C	C	C
	F12	High test file churn.	S		C	
	F13	Test contributors in configurable systems are less productive as a result of variability aspects.	S	S		
	F14	The opposite evolution behavior makes difficult the adoption of similar solutions for SS and configurable systems.		S		
Ownership	F15	High concentration of work.	S	C	C	C
	F16	One-time Contributors occurrence.			S	
Reusable aspects	F17	The scattering and tangle degree are not a problem in the test code.	S			
	F18	There is a correlation between LOC, LOF, and NOFC.	S			
	F19	Variability negatively impacts maintenance.	S			C
	F20	The fraction of variable annotated code in test code is the same as production code.	S	C		
	F21	The interaction in the test code is a great challenge.				S
	F22	The variability difficult the code coverage detection.				S

CS- Case study.
CP- Comparative Study.
ES- Extended Study.
SR- Survey.
S- Source of information.
C- Confirming evidence.
SC- Source of information and Confirming evidence.
CE- Contradictory evidence.

effort in the early stages when the project increased fast due to the initial implementation. This way, the development teams that intend to start developing tests have to consider allocating more contributors at the beginning of the project.

F3 - The effort increases as the project evolve. We notice that the effort increases significantly as the tests evolve, mainly as a result of the growth of the size (LOC) and functional content (Test Cases) (Chapters 4 and 5). Although the effort does not increase after the first quarter of evolution lifetime in Chapter 6, the test contributors' perception confirms the finding of previous chapters. Interestingly, we identified that the number of contributors also rises when considering the ratio between LOC and the number of unique contributors per month. However, the tests exhibit incredibly high values, indicating that the contributors have a great deal of effort to develop and evolve the test suite. This is a concerning situation and needs to be avoided since, as more test files a developer oversees, the more time and effort he will need to devote [25], increasing the time and resources spent in the test activities.

F4 - Contributors' workload is high during evolution. As reported in the **F3**, although the number of contributors rises, the ratio between LOC and the number of unique contributors also increases. This way, we identified that the contributors' workload is increasing. It is a concerning situation since defects in configurable systems generally need a quick fix, and this scenario can make a quick fix harder.

F5 - More contributors have to be only involved with test activities. We identified that the contributors' workload is increasing (**F4**). It is a concerning situation since defects in reusable systems generally need a quick fix, and this scenario can make a quick fix harder. Development teams have to be watchful of this situation, and whether necessary, should consider having more contributors only involved with test activities. This situation is especially true at the beginning of the project when a considerable effort is spent due to the fast growth caused by the initial implementation (**F2**). This action can contribute to a better distribution of the tasks, and consequently, the workload reduction.

F6 - Not all projects include tests at the beginning. In chapter 5, we identified that part of the projects does not have tests at the beginning of the project's lifetime. It was also confirmed in chapter 6, where we observed that 12.7% of the projects take more than ten releases to introduce the tests. This is a concerning situation since the quality of the projects is an essential factor for the stability of reusable systems. The presence of tests, in the beginning, can avoid that defective code propagates errors to a large number of systems. Thus, project developers should include tests since the project start.

F7 - Although there is a perception of quality in reusable systems from a testing perspective, we identified not all projects perform tests. One of the most cited [42] reasons for using projects in reusable systems is the perception of quality. However, we identified that 14.5% of the projects do not have any test script. Therefore, from testing, the perception of the quality does not hold. This situation is concerning since projects that do not have tests still have a lot of clients. These projects without tests become the systems more error-prone. This way, the utilization of projects without tests should be avoided, and development teams need to consider analyze if a project

performs tests and check the history of reported problems. In the short and medium term, the adoption of these steps can avoid unexpected issues.

F8 - Lack of Test framework considering the reusable aspects. Test frameworks are an essential part of any successful automated testing process and can improve the test quality [5]. In Chapters 4 and 5, we observed that the projects do not use any test frameworks considering the variability aspects. In Chapter 6, we identified that 71% of the projects do not use any test framework. This is a concerning situation that results in more effort in the test development. We believed that creating frameworks considering these specifics points of reusable systems could help the development teams control and improve upon their current work processes and productivity and improve the test quality [5].

F9 - The test code maintainability is low over the evolution. In Chapters 4 and 5 we study how some maintainability sub-characteristics of the test evolve. Although not all metrics selected to evaluate the maintainability present unsatisfactory indicators, the overall assessment of test code maintainability is low. We observed that the test code's maintenance is not easy and could become more difficult as the project evolves. Additionally, our results indicate that the variability of existing tests increases the maintainability challenges.

F10 - Some test smells are more prone to occurrence in reusable systems. The analysis of maintainability sub-characteristics show that the test code in reusable systems is more susceptible to the presence of the following test smells:

- **Obscure and Eager Test smell** - The high percentage of test programs with significant values of unit size (LOC) (Chapters 4 and 5) is a warning for the presence of these smells [101]. An obscure test is a test that has a lot of noise in it, making it hard to understand, and consequently, hard to maintain. This test does not serve as documentation, while an eager test attempts to test too much functionality.
- **Conditional Test Logic code smell** - In the studies (Chapters 4 and 5) we identified a high incidence of the test code with a very high levels of unit complexity. The complexity should be kept as low as possible to avoid writing tests for test code [23]. This is also underlined in the description of the Conditional Test Logic code smell [101], which advocates keeping the number of possible paths as low as possible to keep tests correct and straightforward.
- **Fragile test smell** - We observed unit dependence values totally out of the scale in test code (Chapter 4). In the context of test code, the modules' coupling is minimal [23], and high values indicate that the test can be divided into more units. Thus, high unit dependence makes the tests more likely to occur fragile test smell [101], and generally, the presence of this smell increases the test code maintenance effort [101].
- **Test code duplication** - We identified a high percent of cloned functions (Chapters 4 and 5), and the clones are scattered in the files (Chapter 4). Test code duplication

occurs when copy-paste is used as a way to reuse test logic. This results in many copies of the same code, which may significantly increase the test maintenance cost and make the evolution significantly more difficult [23, 105]. It also affects stability since unmanaged code clones can lead to partially applying a change to the clones, thus introducing logical errors in the test code.

Some possible solutions for these smells in the context of SS are proposed in [101]. However, we identified that the opposite evolution behavior (Chapter 5) could make complex the adoption of similar solutions of SS in reusable systems. This way, more investigation about the nature of these smells in configurable systems needs to be performed to define suitable solutions.

F11 - Lack of tool support to the specific context of reusable systems. Our results show that variability increases the challenges in test development and maintenance, and contributors generally do not recognize (potentially harmful) test smells [125]. Additionally, the smell detection process (**F10**) is not too straightforward for configurable systems. This way, tool builders should create or extends existing tools.

F12 - High test file churn. We observed that despite the configurable systems' size and complexity, the addition and exclusion of test cases are frequent. A high changeability is a fact that may significantly increase the test maintenance cost. Although the high value of test file churn suggests that test files are continually evolving [138, 19], this is an alarming situation since prior research shows that frequently changing on code makes them more error-prone [138]. This way, development teams need to be watchful of the changes in test code to avoid the introduction of errors on code since a defect-ridden test suite could not improve the software's quality.

F13 - Test contributors in configurable systems are less productive as a result of variability aspects. By combining evidence from the Chapters 4 and 5, we identified that the test contributors in reusable systems are less productive than SS when considered the LOC by the number of contributors. It can be explained by the presence of variability in the test code, and other aspects, such as feature interaction [108], that make simple alterations in configurable code more time-consuming (**F9**). This way, in the analysis of configurable systems projects, researchers should not only focus on LOC to evaluate the productivity of the test contributors but also consider variability aspects. We believed that the creation of tools considering these specifics points in configurable systems could help the development teams control and improve upon their current work processes and productivity. Also, studies like [126, 127] to better understand productivity and work habits in configurable systems can be very useful.

F14 - The opposite evolution behavior makes difficult the adoption of similar solutions for SS and configurable systems. In chapter 5, we observed that configurable systems and single systems have different evolution behavior along most of the sub-characteristics analyzed. This way, some possible solutions proposed for SS could make it challenging to adopt in configurable systems. We conjecture that where the differences, more investigation about the nature of these differences is needed to define suitable solutions.

F15 - High concentration of work. Chapters 4, and 5 we observed a high concentration of work since 20% of contributors are responsible for 92% of the work performed in the test activities in configurable systems. Recent studies [123, 124] have shown that a high concentration of activities does not necessarily cause bottlenecks in development and communication or imply in the quality. However, test activities have different dynamics [23], and according to the test contributors, a high concentration in the test development is bad in the long term, and it can be an alert related to the quality of code.

F16 - One-time Contributors occurrence. Test contributors need to be watchful of the alterations made by One-time Contributors. In Chapter 6, we identified a high value of One-time Contributors. According to [140], contributors often struggle with code that they are not familiar. However, test and source code tend to co-evolve [11, 23], i.e., changes to the source code often require modifications to the test files, and vice versa, a novice developer may easily introduce a source code change, unaware that a test change may be necessary. If the test is not changed when a change is required, it can cause tests to break (fragile test code smell [101]) and increases the test code maintenance effort [23]. Thus, it is essential the tool support to maps the dependence between the production code and test code to assist the OTC in identifying code changes that require changes in the test suite. Additionally, popular package developers should encourage OTCs to place further contributors.

F17 - The scattering and tangle degree are not a problem in the test code. In Chapter 4, we observed that although some features do not fit well into the architectural model and are scattered across the test code, the proportion of scattered features is nearly constant, which may indicate that it is an evolution parameter actively controlled throughout the evolution. Additionally, the feature expressions present low complexity in terms of tangle degrees. Low complexity is preferable because feature expressions that consist of many feature constants impair program comprehension and complex maintenance. This way, we do not find any indications that these characteristics (measurements) are attractive to evaluate the test code evolution.

F18 - There is a correlation between LOC, LOF, and NOFC. We notice that the variability increases as the size of test code growth, *i.e.*, the NOFC, and LOF metrics increases as a result of LOC growth. Researchers' investigation of this correlation can be valuable information since development teams could use the existing tools for LOC measurement to predict the increase of these variability metrics. The prediction becomes possible that contributors take preventive action to reduce the complexity inherent to the variability and make the maintenance easier.

F19 - Variability negatively impacts maintenance. Additionally, we noticed that the test case life cycle is short, and it is not a good indicator of low maintainability [23]. At the test programs (code files) level, we note that test programs with variability remain less time than test programs without variability. This can be an indicator that the variability influence negatively in maintainability since it introduces more changes.

F20 - The fraction of variable annotated code in the test code is the same as the production code. According to [93], the fraction of cpp-annotated code (LOF/LOC)

in some mid-size software systems exceeds 50% of the codebase. However, large software systems (Freebsd, GCC, Linux, and OpenSolaris) contain a small percentage of variable source code compared to the average. [88] analyzed the variability in 20 open-source systems and reported that these systems have an average of 22% of cpp-annotated lines of code. However, in large systems, the percentage of all code annotated lines varies from 9% to 14%. Even though we had analyzed test code that has a different purpose from the production code analyzed by all the other studies cited here, our results (Chapters 4 and 5 also find a similar proportion of annotated code.

F21 - The interaction in the test code is a great challenge. The main concern highlighted by the contributors is the feature interaction. The contributors state that code complexity and the time devoted to implementing changes in the code are increased due to the interaction of features. Determining the influence of feature interactions on a system's behavior has been a challenging subject for decades [108]. Although tool support has been presented [108], it does not cover the test code.

F22 - The variability difficult the code coverage detection. Code coverage improves fault detection. However, the coverage is not direct in configurable systems due to the variability inside the test artifacts, which increases the possible paths. This way, the test suite's coverage is not correlated with its size since the tests may not identify the variability aspects. Additionally, it is challenging to determine whether the tests are covering the correct code interaction. Although different coverage tool has been proposed [5, 63], these tools do not consider the interaction among features in the analysis.

8.4 FINDINGS OF THE MULTI-METHOD APPROACH

The findings F1 to F22 were based on the evidence identified through the research methods. Also, we identified the findings F23 to F28 concerning the multi-method approach analysis. These last findings contribute to a comprehensive body of knowledge about the multi-method research approach.

F23. The empirical study that found the highest number of findings was the case study (Chapter 4). However, it is valid to highlights that this study was the first one, and some of the findings of this study guide the conjunctions of the next steps.

F24. The empirical study which confirmed the highest number of findings was the comparative study (Chapter 5).

F25. The findings F6 and F13 could not have been established without having access to multiple data sources. In Chapter 5 we observed that some projects do not start with tests. However, at that point, this observation was not investigated. In the extended study, we investigated this observation, and we could confirm the perception. The same occurred in the finding F13 when the LOC by number of contributors was used to evaluate the productivity of them. Therefore, the data source triangulation strategy has been used not only to build confidence for the findings F6 to F13, also to their establishment.

F26. One finding (F3) has been identified as a contradictory finding. Although three different studies indicate one direction, one study showed a different perception of this finding. We believed that for this finding, more empirical studies are required to confirm or refute them.

F27. The findings F7, F14, F16, F17, F18, F21, and F22 have not been confirmed by more than one empirical study. Similarly, these not confirmed findings need more empirical investigations.

8.5 RESEARCH QUESTION ANALYSIS

In this section, we summarize the results based on each research question defined in this work (Chapter 1).

RQ1 - How much effort is required to evolve test suite?

We identified that the effort to develop the test suite is high, mainly in the early stages of the project. Moreover, we observed that few test suites concentrated a significant part of the development effort, and the effort increase as the projects evolve. As a result, the test contributors presented a high workload during evolution. One possible solution to solve this problem could be the adoption of contributors only involved in test activities. Other factors as the high concentration of work and the lack of tool support considering the variability aspects also contribute to the increased effort required to develop and evolve the test suite in reusable systems.

RQ2 - How maintainable is the test suite?

Although, not all the maintainability sub-characteristics investigated present unsatisfactory indicators, the overall assessment of test code maintainability is low. Also, we identified that the maintenance of the project is not easy and could become more problematic as it evolves. We observed that a significant percentage of the test code is classified with a high chance of problems, indicating that tests' maintenance is difficult. Additionally, the presence of high values of cloned functions is a concerning situation since it increases the test code's maintenance effort. Finally, the lack of tool support to the specific context of reusable systems also increases the maintainability.

RQ3 - How is the ownership of test code?

We identified a high concentration of work of the test contributors in configurable systems. Although recent studies have shown that a high concentration of activities does not necessarily cause bottlenecks in development and communication or imply in the quality, the test contributors' perception contrary, and according to them a high concentration is bad in the long term, and it can be an alert related to the quality of code. Additionally, we observed a high value of One-time Contributors. The test code made by OTCs is more prone to introduction of errors, resulting in the increases the test code maintenance effort. Thus, we had identified that the balance of work needs of constant evaluation by the development teams to keep the sustainability of the test as the projects evolve.

RQ4 - Are there reusable aspects that affect the tests?

We observed that the variability negatively impacts maintenance since it introduces more complexity in the test code, consequently more time is devoted to implementing changes. According the test contributors' perception the main challenge is the interaction inside the code. We identified that there is a correlation between LOC, LOF, and NOFC. This correlation could be used by development teams to predict the increase of these variability

metrics. The prediction becomes possible that contributors take preventive action to reduce the complexity inherent to the variability and make the maintenance easier. On the other hand, we identified that the scattering and tangle degree are not a problem in the test code, and this way, these indications are unattractive to evaluate the test code evolution related to variability.

8.6 MULTI-METHOD APPROACH EVALUATION

Applying the multi-method approach to test evolution in the context of configurable systems has produced some exciting and important findings. As a result, we identified some benefits of the approach's application: (i) approach has demonstrated several consistent findings across the phases, i.e., the approach has provided confirmatory power for its results, and, hence, they are more reliable, and (ii) the approach became easier the identification of important hypotheses for further investigation within the research program. As a result of these benefits, the research program focuses on significant and more critical problems. On the other hand, as an original research undertaking, we can cite a drawback: the time and effort required to plan, design, and organize each phase of the research program and analyze the collected data was considerably more than expected.

Although it is argued that the multi-method approach's success far outweighs the shortcomings, there are lessons to be learned. As a follow, we make some recommendations that should aid researchers in attempting similar research programs.

- The high time and effort required to conduct a multi-method research program can be reduced by the best planning of the phases and the inclusion of more people to execute the study if necessary.
- Each phase must be recognized as an empirical study in its own right and not just part of a multi-method approach. This offers two advantages: (i) the details of each phase become available as the multi-method approach evolves, and (ii) each phase should be reported with enough details to be replicated, and the findings should be important enough to warrant it.

8.7 CHAPTER SUMMARY

The multi-method approach allowed us to achieve more reliable and generalized results. Moreover, this approach demonstrated confirmatory power as most findings (facts) had been consistently presented and validated by more than one empirical method. Thus, the software engineering community is more likely to have confidence in the reliability of the findings. Additionally, this approach's application produced empirical results that resulted in a body of knowledge about test evolution in configurable systems.

We believe that the findings of this study can serve as the basis to propose methodologies for test evolution in configurable systems and contribute to the design of new techniques, tools, or frameworks in the future. Additionally, the effort's comprehension can help the community better plan the test development and reduce the costs spent in the test phase.

The next chapter presents the conclusions discussing our contributions, limitations and outlines directions for future work.

CONCLUDING REMARKS AND FUTURE WORK

Configurable Systems allow end-users to customize a system to suit their needs and expected operational context [80]. It is based on the idea that configurable components, built based on a standard design, can be configured and composed in different ways, enabling the creation of a diversity of products [10]. The main benefits of this development strategy are reducing the time to market, as mass-customization facilitates the creation of tailored solutions, and improved software quality, as re-used components are tested in different contexts [60].

Developing a configurable operational system is not easy [10] since these systems have to cope with a considerable number of features that represent a unit of the functionality of a software system and provide a potential configuration option [157]. The features can be shared among several products, and any change in one feature may affect all related products [10]. This way, it is crucial to have a good test suite to ensure product quality and to facilitate future changes. However, developing and maintaining a test suite is time-intensive [5, 7] and costly [8]. According to Brooks [9], the total time devoted to testing is 50% of the total allocated time, while Kung et al. [8] suggest that 40 to 80% of the development costs of software development is spent in the testing phase. This scenario puts software development projects in a difficult situation: on the one hand, tests are essential for software success; on the other hand, tests become a severe burden during maintenance [11].

Interestingly, existing research in configurable systems has focused much of its efforts on variability evolution as it occurs in the variability model. Still, it has ignored the evolution of other related artifacts [75, 76, 77, 78, 79, 80]. The existing studies covering variability evolution across different artifacts focus mainly on production code and build systems [19, 12]. Thus, as an incipient topic, test evolution in configurable systems requires a solid body of knowledge to guide test contributors and development teams to take appropriate procedures for handling test evolution. As an attempt to bridge such a gap, in this thesis, we employed a multi-method approach to developing an in-depth understanding of test evolution in configurable systems to unveil evidence on the topic from a range of sources.

We emphasize that empirical evidence can promote the aimed understanding based on factors that lead to test evolution from the point of view of effort and maintenance and mitigation strategies to address these factors, building further knowledge about test evolution. From that knowledge, it can be easier to develop tools to support the test evolution process and methodologies that help the development teams to maintain the test suite.

Along this Chapter, we summarize the main research contributions (Section 9.1), discuss a future agenda, based on the set of yet-to-solve topics (Section 9.2), and draw concluding remarks (Section 9.3).

9.1 SUMMARY OF CONTRIBUTIONS

Some test evolution studies were identified through the informal literature review study described in Chapter 3. Although we do not find studies focusing on test artifacts as the systems evolve, the key difference between this work and the previous ones is that we combine different but complementary studies to address research. Additionally, we performed a research synthesis that can provide both researchers and practitioners with a means to define guidelines and adopt quality procedures to address test evolution issues. Next, we present the contributions of this Thesis.

This work's main contributions can be split into the following aspects: (i) a set of empirical studies combining evidence from different sources, (ii) a set of empirical findings to achieve the understanding of the test evolution, and (iii) dataset for the software engineering community. These contributions are further described next.

- ***A set of empirical studies.*** Set of empirical studies that combine evidence from different sources and to achieve the research's objective. In the first phase of the research program (Chapter 3), we provided an overview of the existing research related to this thesis's subjects and presented related work to our investigation. The goal was to map out the test evolution field in reusable systems, synthesize available evidence to suggest important implications for practice, and identify research trends, open issues, and areas for improvements. The second phase was composed of four empirical studies. First, we performed a case study to analyze the test evolution of a large configurable system. Second, we conducted a comparative study to evaluate the test evolution in 18 open-source projects from various sizes and domains in configurable systems and their similarities and differences to 18 SS projects. The comparison allows us to identify similarities that can help us adapt techniques from SS to configurable systems and vice versa. The differences can help us design new strategies addressing the discrepancies. Third, we performed a comprehensive study to analyze the test suite evolution in another category of reusable systems to verify whether some observations are recurring and gather new data that support the findings. Finally, we surveyed test contributors to investigate the test evolution from the development point of view and improve the conclusions in previous stages. The survey study extracted evidence from experts and built a perception of reality compared to the evidence extracted in earlier studies. In general, the evidence extracted during previous studies was confirmed, and new evidence was identified.

- ***A set of empirical findings.*** The set of empirical findings from this multi-method research approach promoted the understanding of the test evolution in configurable systems in many perspectives, such as: (i) evidence of the effort to evolve the test suite; (ii) evidence related to maintainability sub-characteristics of the test evolution; (iii) variability aspects that impacts in the test evolution; and (iv) analysis about the human factors that may influence the test evolution. The knowledge base from these findings makes the test evolution in configurable systems easier and reduces the time and cost to evolve the tests.
- ***Dataset.*** By mining the versioned systems of open source projects from various sizes and domains used in the studies on this thesis, we provided three large datasets on test evolution. The dataset can provide researchers with a testbed that can be used for empirical research in software engineering;

9.2 FUTURE WORK

This work is an achievement towards a guide for test evolution in configurable systems, and exciting directions remain to improve what was started here, and new routes can be explored in the future. Thus, the following issues should be investigated as future work:

- ***Model of maintainability for test code in configurable systems*** - This thesis presented the first investigations of the maintainability of test code in configurable systems in an extended period. To evaluate the maintenance, we used the maintainability sub-characteristics from the test code quality model proposed by [23]. Future research includes investigating the relevance of the chosen metrics for maintainability in configurable systems' test code and formal proposition and validation of a test evolution model in the context of variability.
- ***Dynamic analysis of the test code*** - We only provided a static analysis of the test suite, and dynamics aspects are not considered in the evaluation. Future work can include dynamics analyses such as code coverage, and fault detection effectiveness of test suite in configurable systems [99].
- ***Tool support*** - Tool support has been out of the scope of this research. However, developing a new tool or extending an existing one for supporting the test evolution process will be an essential contribution to this area.
- ***More Empirical Studies*** - This Thesis presented the definition, planning, operation, analysis, interpretation, and packaging of a set of empirical studies. However, new studies in different contexts, including various subjects and other domains, are still necessary to increment the collection of empirical evidence extracted.

9.3 CONCLUDING REMARKS

Configurable systems have to cope with considerable numbers of points of variabilities. The high degree of variability and features spread across many software artifacts makes

variability pervasive in the system. This way, it is of utmost importance to have a test suite available to ensure the current state of the software system and ease future changes. Surprisingly, little is known about test evolution in configurable systems, directly impacting the quality of existing tools and methodologies.

This work presented a set of empirical evidence that aims to promote the understanding of test evolution in configurable systems towards advancing the research concerning test evolution. Based on the identified reasons that increase the test suite effort and maintainability, we provided some implications for development teams and researchers. Such implications can improve the current practices, tools and make the software tests less expensive and time-consuming.

Additionally, this Thesis presented a multi-method approach used to perform empirical software engineering research. The multi-method research methodology was adopted in this investigation to understand better how the tests evolve in configurable systems. According to the data collected and analyzed, the approach presented indications of its viability. We believe that this Thesis is one more step to the maturation of the test evolution topic.

BIBLIOGRAPHY

- 1 LEHMAN, M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, v. 1, p. 213 – 221, 1979. ISSN 0164-1212.
- 2 Lehman, M. M.; Ramil, J. F.; Wernick, P. D.; Perry, D. E.; Turski, W. M. Metrics and laws of software evolution-the nineties view. In: *Proceedings Fourth International Software Metrics Symposium*. [S.l.: s.n.], 1997. p. 20–32.
- 3 LEHMAN, M. m. Software’s future: Managing evolution. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 15, n. 1, p. 40–44, jan. 1998. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/MS.1998.646878>.
- 4 Bennett, K. Legacy systems: coping with success. *IEEE Software*, v. 12, n. 1, p. 19–23, Jan 1995.
- 5 MOONEN, L.; DEURSEN, A. van; ZAIDMAN, A.; BRUNTINK, M. On the interplay between software testing and evolution and its effect on program comprehension. In: _____. *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 173–202. ISBN 978-3-540-76440-3.
- 6 Reiss, S. P. Constraining software evolution. In: *International Conference on Software Maintenance, 2002. Proceedings*. [S.l.: s.n.], 2002. p. 162–171.
- 7 ELLIMS, M.; BRIDGES, J.; INCE, D. C. The economics of unit testing. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 11, n. 1, p. 5–31, mar. 2006. ISSN 1382-3256.
- 8 KUNG, D. C.; GAO, J.; KUNG, C.-H. *Testing Object-Oriented Software*. 1st. ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998. ISBN 0818685204.
- 9 BROOKS JR., F. P. *The Mythical Man-month (Anniversary Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-83595-9.
- 10 MACHADO, I. D. C.; MCGREGOR, J. D.; CAVALCANTI, Y. a. C.; ALMEIDA, E. S. D. On strategies for testing software product lines: A systematic literature review. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 56, n. 10, p. 1183–1199, out. 2014. ISSN 0950-5849.
- 11 ZAIDMAN, A.; ROMPAEY, B. V.; DEMEYER, S.; DEURSEN, A. v. Mining software repositories to study co-evolution of production & test code. In: *2008 1st International Conference on Software Testing, Verification, and Validation*. [S.l.: s.n.], 2008. p. 220–229. ISSN 2159-4848.

- 12 ZAIDMAN, A.; ROMPAEY, B.; DEURSEN, A.; DEMEYER, S. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 16, n. 3, p. 325–364, jun. 2011. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-010-9143-7>.
- 13 PINTO, L. S.; SINHA, S.; ORSO, A. Understanding myths and realities of test-suite evolution. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2012. (FSE '12), p. 33:1–33:11. ISBN 978-1-4503-1614-9.
- 14 PINTO, L. S.; SINHA, S.; ORSO, A. TestEvol: A tool for analyzing test-suite evolution. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.]: IEEE, 2013. p. 1303–1306. ISBN 978-1-4673-3076-3. ISSN 02705257.
- 15 HILTON, M.; BELL, J.; MARINOV, D. A large-scale study of test coverage evolution. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2018. (ASE 2018), p. 53–63. ISBN 978-1-4503-5937-5.
- 16 SOUTO, S.; D'AMORIM, M.; GHEYI, R. Balancing soundness and efficiency for practical testing of configurable systems. In: *Proceedings of the 39th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE '17), p. 632–642. ISBN 978-1-5386-3868-2.
- 17 Kim, C. H. P.; Khurshid, S.; Batory, D. Shared execution for efficiently testing product lines. In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2012. p. 221–230. ISSN 1071-9458.
- 18 NGUYEN, H. V.; KÄSTNER, C.; NGUYEN, T. N. Exploring variability-aware execution for testing plugin-based web applications. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 907–918. ISBN 978-1-4503-2756-5.
- 19 MCINTOSH, S.; ADAMS, B.; NGUYEN, T. H.; KAMEI, Y.; HASSAN, A. E. An empirical study of build maintenance effort. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, p. 141, 2011. ISSN 0270-5257.
- 20 MCINTOSH, S. Build system maintenance. n. Section 4, p. 1167, 2011.
- 21 NETO, C. R. L.; NETO, P. A. da M. S.; ALMEIDA, E. S. de; MEIRA, S. R. de L. A mapping study on software product lines testing tools. In: *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE'2012), Hotel Sofitel, Redwood City, San Francisco Bay, USA July 1-3, 2012*. [S.l.]: Knowledge Systems Institute Graduate School, 2012. p. 628–634.

- 22 Kudrjavets, G.; Nagappan, N.; Ball, T. Assessing the relationship between software assertions and faults: An empirical investigation. In: *2006 17th International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2006. p. 204–212. ISSN 1071-9458.
- 23 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, Nov 2014. ISSN 0098-5589.
- 24 BIRD, C.; NAGAPPAN, N.; MURPHY, B.; GALL, H.; DEVANBU, P. Don't touch my code! examining the effects of ownership on software quality. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 4–14. ISBN 9781450304436.
- 25 ZHOU, M.; CHEN, Q.; MOCKUS, A.; WU, F. On the scalability of linux kernel maintainers' work. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 27–37. ISBN 978-1-4503-5105-8. Disponível em: <http://doi.acm.org/10.1145/3106237.3106287>.
- 26 RUNESON, P.; HÖST, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 14, p. 131–164, April 2009. ISSN 1382-3256.
- 27 WOOD, M.; DALY, J.; MILLER, J.; ROPER, M. Multi-method research: an empirical investigation of object-oriented technology. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 48, p. 13–26, August 1999. ISSN 0164-1212.
- 28 BROOKS, A.; DALY, J.; MILLER, J.; ROPER, M.; WOOD, M. *Replication of Experimental Results in Software Engineering*. [S.l.], 1996.
- 29 BREWER, J.; HUNTER, A. *Multimethod Research: A Synthesis of Styles*. [S.l.]: Sage Publications, 1989. 1–5 p.
- 30 MARTIN, J. A garbage can model of the research process. *Judgment calls in research*, Sage, Book, p. 17–39, 1982.
- 31 MARINESCU, P. D.; HOSEK, P.; CADAR, C. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In: *International Symposium on Software Testing and Analysis, ISSSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. [S.l.: s.n.], 2014. p. 93–104.
- 32 COOPER, D.; SCHINDLER, P. *Business Research Methods*. [S.l.]: Mcgraw-Hill College; 8th edition, 1998.
- 33 GILL, J.; JOHNSON, P. *Research Methods for Managers*. [S.l.]: London: Paul, 1991.
- 34 GALLIERS, R. *Choosing Information System Research Approaches*. [S.l.]: Information System Research: issues, methods and practical guidelines, 1992.

- 35 FRANZ C.R., R. D.; KOEBLITZ, R. User response to an online information system: A field experiment. *MIS Quarterly*, v. 10, p. 29–42, 1986.
- 36 MORSE, J. M. Principles of mixed methods and multimethod research designs. In *A. Tashakkori & C. Teddlie (Eds.) Handbook of mixed methods in social & behavioral research*, p. 189–208, 2003.
- 37 MIXED Methods Research: Merging Theory with Practice. *Qualitative Social Work*, v. 11, n. 2, p. 220–225, 2012. Disponível em: <https://doi.org/10.1177/1473325011433761b>.
- 38 BASTOS, J. F.; NETO, P. A. da M. S.; ALMEIDA, E. S. de; MEIRA, S. R. de L. Software product lines adoption: An industrial case study (keynote). In: *Proceedings of the Third International Workshop on Conducting Empirical Studies in Industry*. Piscataway, NJ, USA: IEEE Press, 2015. (CESI '15), p. 35–42.
- 39 BASTOS, J. F.; NETO, P. A. da M. S.; OLEARY, P.; ALMEIDA, E. S. de; MEIRA, S. R. de L. Software product lines adoption in small organizations. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 131, n. C, p. 112–128, set. 2017. ISSN 0164-1212. Disponível em: <https://doi.org/10.1016/j.jss.2017.05.052>.
- 40 NETO, P. A. da M. S.; MACHADO, I. d. C.; MCGREGOR, J. D.; ALMEIDA, E. S. de; MEIRA, S. R. de L. A systematic mapping study of software product lines testing. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 53, n. 5, p. 407–423, maio 2011. ISSN 0950-5849.
- 41 GAROUSI, V.; KüçüK, B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, v. 138, p. 52 – 81, 2018. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121217303060>.
- 42 ABDALKAREEM, R.; NOURRY, O.; WEHAIBI, S.; MUJAHID, S.; SHIHAB, E. Why do developers use trivial packages? an empirical case study on npm. In: *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. [S.l.: s.n.], 2017. p. 385–395. ISBN 9781450351058.
- 43 POHL, K.; BöCKLE, G.; LINDEN, F. J. v. d. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540243720.
- 44 ALMEIDA, E. S. de. Software reuse and product line engineering. In: _____. *Handbook of Software Engineering*. Cham: Springer International Publishing, 2019. p. 321–348. ISBN 978-3-030-00262-6.
- 45 TRACZ, W. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63369-8.

- 46 BASILI, V. R.; BRIAND, L. C.; MELO, W. L. How reuse influences productivity in object-oriented systems. *Commun. ACM*, ACM, New York, NY, USA, v. 39, n. 10, p. 104–116, out. 1996. ISSN 0001-0782.
- 47 EZRAN, M.; MORISIO, M.; TULLY, C. *Practical Software Reuse*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 1-85233-502-5.
- 48 KRUEGER, C. W. Software reuse. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 24, n. 2, p. 131–183, jun. 1992. ISSN 0360-0300.
- 49 POHL, K.; METZGER, A. Variability management in software product line engineering. In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006. (ICSE '06), p. 1049–1050. ISBN 1-59593-375-1.
- 50 APEL, S.; KÄSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology*, v. 8, n. 5, p. 49–84, jul 2009. ISSN 1660-1769.
- 51 SOARES, L. R.; MACHADO, I. do C.; ALMEIDA, E. S. de. Non-functional properties in software product lines: A reuse approach. In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*. New York, NY, USA: ACM, 2015. (VaMoS '15), p. 67:67–67:74. ISBN 978-1-4503-3273-6.
- 52 CZARNECKI, K.; EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.
- 53 Medeiros, F.; Ribeiro, M.; Gheyi, R.; Apel, S.; Kästner, C.; Ferreira, B.; Carvalho, L.; Fonseca, B. Discipline matters: Refactoring of preprocessor directives in theifdfhell. *IEEE Transactions on Software Engineering*, v. 44, n. 5, p. 453–469, May 2018. ISSN 0098-5589.
- 54 BOSCH, J. From software product lines to software ecosystems. In: *Proceedings of the 13th International Software Product Line Conference*. Pittsburgh, PA, USA: Carnegie Mellon University, 2009. (SPLC '09), p. 111–119.
- 55 MCGREGOR, J. D. Testing a software product line. In: _____. *Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 104–140. ISBN 978-3-642-14335-9.
- 56 PATTON, R. *Software Testing (2Nd Edition)*. Indianapolis, IN, USA: Sams, 2005. ISBN 0672327988.
- 57 COLLARD, J.-F.; BURNSTEIN, I. *Practical Software Testing*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 0387951318.
- 58 AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381.

- 59 MYERS, G. J.; SANDLER, C. *The Art of Software Testing*. USA: John Wiley & Sons, Inc., 2004. ISBN 0471469122.
- 60 CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001. ISBN 0201703327.
- 61 MONTALVILLO, L.; DÍAZ, O. Requirement-driven evolution in software product lines. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 122, n. C, p. 110–143, dez. 2016. ISSN 0164-1212.
- 62 HARROLD, M. J. Testing: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 61–72. ISBN 1-58113-253-0.
- 63 HILTON, M.; BELL, J.; MARINOV, D. A large-scale study of test coverage evolution. In: KASTNER, C.; HUCHARD, M.; FRASER, G. (Ed.). *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. [S.l.]: Association for Computing Machinery, Inc, 2018. p. 53–63.
- 64 ELBAUM, S. G.; GABLE, D.; ROTHERMEL, G. The impact of software evolution on code coverage information. In: *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*. [S.l.: s.n.], 2001. p. 170–179.
- 65 MIRZAAGHAEI, M.; PASTORE, F.; PEZZÈ, M. Automatic test case evolution. *Software Testing, Verification and Reliability*, v. 24, n. 5, p. 386–411, 2014.
- 66 ENGSTRÖM, E.; RUNESON, P. Software product line testing - A systematic mapping study. *Information & Software Technology*, v. 53, n. 1, p. 2–13, 2011.
- 67 MEDEIROS, F.; KÄSTNER, C.; RIBEIRO, M.; GHEYI, R.; APEL, S. A comparison of 10 sampling algorithms for configurable systems. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 643–654. ISBN 978-1-4503-3900-1.
- 68 SOUTO, S.; D'AMORIM, M. Time-space efficient regression testing for configurable systems. *Journal of Systems and Software*, v. 137, p. 733–746, 2018.
- 69 CHRISTIAN, K.; HERBSLEB, J.; BOGART, C.; KÄSTNER, C.; HERBSLEB, J.; THUNG, F. How to break an API : Cost negotiation and community values in three software ecosystems. In: *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2016. p. 13–18. ISBN 9781450342186.
- 70 DECAN, A.; MENS, T.; CONSTANTINOU, E. On the impact of security vulnerabilities in the npm package dependency network. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. [S.l.: s.n.], 2018. p. 181–191.

- 71 AXELSSON, J.; SKOGLUND, M. Quality assurance in software ecosystems: A systematic literature mapping and research agenda. *Journal of Systems and Software*, v. 114, p. 69 – 81, 2016. ISSN 0164-1212.
- 72 GREILER, M.; DEURSEN, A. van. What your plug-in test suites really test: an integration perspective on test suite understanding. *Empirical Software Engineering*, v. 18, n. 5, p. 859–900, Oct 2013. ISSN 1573-7616.
- 73 KULA, R. G.; GERMAN, D. M.; ISHIO, T.; INOUE, K. Trusting a library: A study of the latency to adopt the latest Maven release. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. [S.l.: s.n.], 2015. p. 520–524. ISBN 9781479984695. ISSN 1534-5351.
- 74 Claes, M.; Mens, T.; Grosjean, P. On the maintainability of cran packages. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. [S.l.: s.n.], 2014. p. 308–312.
- 75 ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P.; LUCENA, C. Refactoring product lines. In: . [S.l.: s.n.], 2006. p. 201–210.
- 76 SHE, S.; LOTUFO, R.; BERGER, T.; ASOWSKI, A. W.; CZARNECKI, K. The variability model of the linux kernel. In: *In VaMoS*. [S.l.: s.n.], 2010.
- 77 LOTUFO, R.; SHE, S.; BERGER, T.; CZARNECKI, K.; WASOWSKI, A. Evolution of the linux kernel variability model. In: *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*. Berlin, Heidelberg: Springer-Verlag, 2010. (SPLC'10), p. 136–150. ISBN 3-642-15578-2, 978-3-642-15578-9.
- 78 GUO, J.; WANG, Y.; TRINIDAD, P.; BENAVIDES, D. Consistency maintenance for evolving feature models. *Expert Syst. Appl.*, v. 39, p. 4987–4998, 04 2012.
- 79 PASSOS, L. T.; TEIXEIRA, L.; DINTZNER, N.; APEL, S.; WASOWSKI, A.; CZARNECKI, K.; BORBA, P.; GUO, J. Coevolution of variability models and related software artifacts - A fresh look at evolution patterns in the linux kernel. *Empirical Software Engineering*, v. 21, n. 4, p. 1744–1793, 2016.
- 80 DINTZNER, N.; DEURSEN, A. van; PINZGER, M. FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering*, v. 23, n. 2, p. 905–952, 2018.
- 81 LARSON, P.; HINDS, N.; RAVINDRAN, R.; FRANKE, H. *Improving the Linux Test Project with Kernel Code Coverage Analysis*. [S.l.], 2003.
- 82 LARSON, P. Testing linux with the linux test project. In: *Proceedings of the Ottawa Linux Symposium*. [S.l.: s.n.], 2002. p. 265–273. ISBN 1-58113-253-0.
- 83 TORVALDS, L. The linux edge. *Commun. ACM*, ACM, New York, NY, USA, v. 42, n. 4, p. 38–39, abr. 1999. ISSN 0001-0782.

- 84 BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: *Encyclopedia of Software Engineering*. [S.l.]: Wiley, 1994.
- 85 MODAK, S.; SINGHM, B.; YAMATO, M. Putting ltp to test—validating both the linux kernel and test-cases. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2009. p. 209–220.
- 86 NETO, P. A. d. M. S.; MACHADO, I. d. C.; CAVALCANTI, Y. C.; ALMEIDA, E. S. d.; GARCIA, V. C.; MEIRA, S. R. d. L. A regression testing approach for software product lines architectures. In: *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*. [S.l.: s.n.], 2010. p. 41–50.
- 87 HRUBIS, C. *Kernel test automation with LTP*. 2014. <https://lwn.net/Articles/625969/>. Accessed: 2017-01-12.
- 88 HUNSEN, C.; ZHANG, B.; SIEGMUND, J.; KÄSTNER, C.; LEBENICH, O.; BECKER, M.; APEL, S. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 2, p. 449–482, abr. 2016. ISSN 1382-3256.
- 89 EL-SHARKAWY, S.; YAMAGISHI-EICHLER, N.; SCHMID, K. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, v. 106, p. 1 – 30, 2019. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584918301873>.
- 90 BAGGEN, R.; CORREIA, J. P.; SCHILL, K.; VISSER, J. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, v. 20, n. 2, p. 287–307, Jun 2012. ISSN 1573-1367. Disponível em: <https://doi.org/10.1007/s11219-011-9144-9>.
- 91 PASSOS, L.; GUO, J.; TEIXEIRA, L.; CZARNECKI, K.; WASOWSKI, A.; BORBA, P. Coevolution of variability models and related artifacts: A case study from the linux kernel. In: *Proceedings of the 17th International Software Product Line Conference*. New York, NY, USA: ACM, 2013. (SPLC '13), p. 91–100. ISBN 978-1-4503-1968-3.
- 92 PASSOS, L.; PADILLA, J.; BERGER, T.; APEL, S.; CZARNECKI, K.; VALENTE, M. T. Feature scattering in the large: A longitudinal study of linux kernel device drivers. In: *Proceedings of the 14th International Conference on Modularity*. New York, NY, USA: ACM, 2015. (MODULARITY 2015), p. 81–92. ISBN 978-1-4503-3249-1.
- 93 LIEBIG, J.; APEL, S.; LENGAUER, C.; KÄSTNER, C.; SCHULZE, M. An analysis of the variability in forty preprocessor-based software product lines. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 105–114. ISBN 978-1-60558-719-6.
- 94 LIEBIG, J.; KÄSTNER, C.; APEL, S. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: *Proceedings of the Tenth International Conference*

- on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2011. (AOSD '11), p. 191–202. ISBN 978-1-4503-0605-8.
- 95 SHIHAB, E.; KAMEI, Y.; ADAMS, B.; HASSAN, A. E. Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, v. 55, n. 11, p. 1981 – 1993, 2013. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584913001316>.
- 96 COOK, S.; HARRISON, R.; WERNICK, P. A simulation model of self-organising evolvability in software systems. In: *IEEE International Workshop on Software Evolvability (Software-Evolvability'05)*. [S.l.: s.n.], 2005. p. 17–22.
- 97 CAPILUPPI, A.; MORISIO, M.; RAMIL, J. F. Structural evolution of an open source system: a case study. In: *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004*. [S.l.: s.n.], 2004. p. 172–182. ISSN 1092-8138.
- 98 N CAPILUPPI A, R. J. S. A study of open source software evolution data using qualitative simulation. In: *Software Process Improvement and Practice*. [S.l.: s.n.], 2005. p. 287—300.
- 99 ISRAELI, A.; FEITELSON, D. G. The linux kernel as a case study in software evolution. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 83, n. 3, p. 485–501, mar. 2010. ISSN 0164-1212.
- 100 JONES, C.; BONSIGNOUR, O. *The Economics of Software Quality*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 0132582201, 9780132582209.
- 101 MESZAROS, G. *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006. ISBN 0131495054.
- 102 MCCABE, T. J. A complexity measure. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 2, n. 4, p. 308–320, jul. 1976. ISSN 0098-5589.
- 103 MYERS, G. J. An extension to the cyclomatic measure of program complexity. *SIG-PLAN Not.*, ACM, New York, NY, USA, v. 12, n. 10, p. 61–64, out. 1977. ISSN 0362-1340.
- 104 SHEPPERD, M. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, Michael Faraday House, Herts, UK, UK, v. 3, n. 2, p. 30–36, mar. 1988. ISSN 0268-6961.
- 105 Cordy, J. R.; Roy, C. K. The nicad clone detector. In: *2011 IEEE 19th International Conference on Program Comprehension*. [S.l.: s.n.], 2011. p. 219–220. ISSN 1092-8138.
- 106 QUEIROZ, R.; PASSOS, L.; VALENTE, M. T.; APEL, S.; CZARNECKI, K. Does feature scattering follow power-law distributions?: An investigation of five pre-processor-based systems. In: *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2014. (FOSD '14), p. 23–29. ISBN 978-1-4503-2980-4.

- 107 ZHANG, B.; BECKER, M.; PATZKE, T.; SIERSZECKI, K.; SAVOLAINEN, J. E. Variability evolution and erosion in industrial product lines: A case study. In: *Proceedings of the 17th International Software Product Line Conference*. New York, NY, USA: ACM, 2013. (SPLC '13), p. 168–177. ISBN 978-1-4503-1968-3.
- 108 SOARES, L. R.; SCHOBENS, P.-Y.; MACHADO, I. do C.; ALMEIDA, E. S. de. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology*, v. 98, p. 44 – 58, 2018. ISSN 0950-5849.
- 109 LEHMAN, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 1, p. 213–221, set. 1984. ISSN 0164-1212.
- 110 LEGUNSEN, O.; HARIRI, F.; SHI, A.; LU, Y.; ZHANG, L.; MARINOV, D. An extensive study of static regression test selection in modern software evolution. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016. (FSE 2016), p. 583–594. ISBN 978-1-4503-4218-6. Disponível em: <http://doi.acm.org/10.1145/2950290.2950361>.
- 111 LOPEZ-HERREJON, R. E.; FERRER, J.; CHICANO, F.; EGYED, A.; ALBA, E. Evolutionary computation for software product line testing: An overview and open challenges. In: _____. *Computational Intelligence and Quantitative Software Engineering*. Cham: Springer International Publishing, 2016. p. 59–87. ISBN 978-3-319-25964-2.
- 112 ENGSTRÅM, E.; RUNESON, P.; SKOGLUND, M. A systematic review on regression test selection techniques. *Information and Software Technology*, v. 52, n. 1, p. 14 – 30, 2010. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584909001219>.
- 113 BORGES, H.; Tulio Valente, M. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, Elsevier Inc., v. 146, p. 112–129, 2018. ISSN 01641212.
- 114 Izurieta, C.; Bieman, J. M. How software designs decay: A pilot study of pattern evolution. In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. [S.l.: s.n.], 2007. p. 449–451. ISSN 1949-3770.
- 115 Izurieta, C.; Bieman, J. M. Testing consequences of grime buildup in object oriented design patterns. In: *2008 1st International Conference on Software Testing, Verification, and Validation*. [S.l.: s.n.], 2008. p. 171–179. ISSN 2159-4848.
- 116 AHMED, I.; MANNAN, U. A.; GOPINATH, R.; JENSEN, C. An Empirical Study of Design Degradation: How Software Projects Get Worse over Time. *International Symposium on Empirical Software Engineering and Measurement*, v. 2015-Novem, p. 31–40, 2015. ISSN 19493789.

- 117 EL-SHARKAWY, S.; KRAFCZYK, A.; SCHMID, K. Metric haven — more than 23,000 metrics for measuring quality attributes of software product lines. In: *Proceedings of the 23rd International Systems and Software Product Line Conference*. New York, NY, USA: ACM. Accepted.
- 118 CASALNUOVO, C.; DEVANBU, P.; OLIVEIRA, A.; FILKOV, V.; RAY, B. Assert use in github projects. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 755–766. ISBN 978-1-4799-1934-5.
- 119 BAUER, D. F. Constructing confidence sets using rank statistics. *Journal of the American Statistical Association*, Taylor Francis, v. 67, n. 339, p. 687–690, 1972.
- 120 CLIFF, N. *Ordinal Methods for Behavioral Data Analysis*. [S.l.]: Psychology Press, 2014.
- 121 ROMANO, J.; KROMREY, J. D. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? 01 2006.
- 122 ALALFI, M. H.; ANTONY, E. P.; CORDY, J. R. An approach to clone detection in sequence diagrams and its application to security analysis. *Software & Systems Modeling*, v. 17, n. 4, p. 1287–1309, Oct 2018.
- 123 MAJUMDER, S.; CHAKRABORTY, J.; AGRAWAL, A.; MENZIES, T. Why Software Projects need Heroes (Lessons Learned from 1100+ Projects). *IEEE Transactions on Software Engineering*, p. 1–12, 2019.
- 124 AGRAWAL, A.; RAHMAN, A.; KRISHNA, R.; SOBRAN, A.; MENZIES, T. We don't need another hero?: the impact of "heroes" on software development. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. [S.l.: s.n.], 2018. p. 245–253.
- 125 TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016. (ASE 2016), p. 4–15. ISBN 978-1-4503-3845-5.
- 126 MEYER, A. N.; BARTON, L. E.; MURPHY, G. C.; ZIMMERMANN, T.; FRITZ, T. The work life of developers: Activities, switches and perceived productivity. *IEEE Trans. Software Eng.*, v. 43, n. 12, p. 1178–1193, 2017.
- 127 MELO, J.; NARCIZO, F. B.; HANSEN, D. W.; BRABRAND, C.; WASOWSKI, A. Variability through the eyes of the programmer. In: *Proceedings of the 25th International Conference on Program Comprehension*. [S.l.: s.n.], 2017. (ICPC '17), p. 34–44. ISBN 978-1-5386-0535-6.

- 128 BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A.; BINKLEY, D. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, v. 20, n. 4, p. 1052–1094, ago. 2015. ISSN 1382-3256.
- 129 KRÜGER, J.; AL-HAJJAJI, M.; SCHULZE, S.; SAAKE, G.; LEICH, T. Towards automated test refactoring for software product lines. In: *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1*. [S.l.: s.n.], 2018. (SPLC '18), p. 143–148. ISBN 978-1-4503-6464-5.
- 130 BELLON, S.; KOSCHKE, R.; ANTONIOL, G.; KRINKE, J.; MERLO, E. Comparison and evaluation of clone detection tools. *IEEE Trans on Software Engineering*, v. 33, n. 9, p. 577–591, set. 2007. ISSN 0098-5589.
- 131 SAINI, V.; FARMAHINIFARAHANI, F.; LU, Y.; BALDI, P.; LOPES, C. V. Oreo: detection of clones in the twilight zone. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. [S.l.: s.n.], 2018. p. 354–365.
- 132 ROY, C. K.; CORDY, J. R. Benchmarks for software clone detection: A ten-year retrospective. In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. [S.l.: s.n.], 2018. p. 26–37.
- 133 FARMAHINIFARAHANI, F.; SAINI, V.; YANG, D.; SAJNANI, H.; LOPES, C. V. On precision of code clone detection tools. In: *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. [S.l.: s.n.], 2019. p. 84–94.
- 134 SCHULZE, S.; APEL, S.; KÄSTNER, C. Code clones in feature-oriented software product lines. In: *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*. [S.l.: s.n.], 2010. p. 103–112.
- 135 MENS, T.; GOEMINNE, M. Analysing the evolution of social aspects of open source software ecosystems. In: . [S.l.: s.n.], 2011. v. 746, p. 1–14.
- 136 GOUSIOS, G.; ZAIDMAN, A. A dataset for pull-based development research. In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. [S.l.: s.n.], 2014. p. 368–371.
- 137 ZHU, J.; ZHOU, M.; MOCKUS, A. Patterns of folder use and project popularity: a case study of github repositories. In: *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*. [S.l.: s.n.], 2014. p. 30:1–30:4.

- 138 Nagappan, N.; Ball, T. Use of relative code churn measures to predict system defect density. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. [S.l.: s.n.], 2005. p. 284–292. ISSN 0270-5257.
- 139 TANTITHAMTHAVORN, C.; MCINTOSH, S.; HASSAN, A. E.; MATSUMOTO, K. The impact of automated parameter optimization on defect prediction models. *IEEE Trans. Software Eng.*, v. 45, n. 7, p. 683–711, 2019.
- 140 ANTONIOL, G.; GUÉHÉNEUC, Y. Feature identification: A novel approach and a case study. In: *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. [S.l.: s.n.], 2005. p. 357–366.
- 141 KITCHENHAM, B. A.; PFLEEGER, S. L. *Personal Opinion Surveys*. [S.l.]: Springer London, 2008. 63-92 p.
- 142 LI, M.; SMIDTS, C. A ranking of software engineering measures based on expert opinion. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 29, p. 811–824, September 2003. ISSN 0098-5589.
- 143 JHA, M.; O'BRIEN, L. Identifying issues and concerns in software reuse in software product lines. In: *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 181–190.
- 144 AHMED, F.; CAPRETZ, L. F.; SHEIKH, S. A. Institutionalization of software product line: An empirical investigation of key organizational factors. *Journal System Software*, Elsevier Science Inc., New York, NY, USA, v. 80, p. 836–849, June 2007. ISSN 0164-1212.
- 145 Daka, E.; Fraser, G. A survey on unit testing practices and problems. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2014. p. 201–211.
- 146 Kochhar, P. S.; Thung, F.; Nagappan, N.; Zimmermann, T.; Lo, D. Understanding the test automation culture of app developers. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2015. p. 1–10.
- 147 FINK, A. *The Survey Handbook*. [S.l.]: Sage Publications, 2003. 129 p.
- 148 SINGER, J.; SIM, S. E.; LETHBRIDGE, T. C. Software engineering data collection for field studies. In: _____. *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008. p. 9–34. ISBN 978-1-84800-044-5.
- 149 COOKE, R. *Experts in uncertainty: opinion and subjective probability in science*. [S.l.]: Oxford University Press, USA, 1991. 336 p.
- 150 CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 20, p. 476–493, June 1994. ISSN 0098-5589.

- 151 KRÜGER, J.; BERGER, T. An empirical analysis of the costs of clone- and platform-oriented software reuse. In: *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2020. (ESEC/FSE). Accepted.
- 152 KRÜGER, J.; BERGER, T. Activities and costs of re-engineering cloned variants into an integrated platform. In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: Association for Computing Machinery, 2020. (VAMOS '20). ISBN 9781450375016.
- 153 PORTER, A. A.; JOHNSON, P. M. Assessing software review meetings: Results of a comparative analysis of two experimental studies. *IEEE Transactions on Software Engineering*, v. 23, p. 129–145, 1997.
- 154 CRUZES, D. S.; DYBA, T. Research synthesis in software engineering: A tertiary study. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 53, p. 440–455, May 2011. ISSN 0950-5849.
- 155 RODGERS, M.; SOWDEN, A.; PETTICREW, M.; ARAI, L.; ROBERTS, H.; BRITTEN, N.; POPAY, J. Testing Methodological Guidance on the Conduct of Narrative Synthesis in Systematic Reviews: Effectiveness of Interventions to Promote Smoke Alarm Ownership and Function. *Evaluation*, v. 15, n. 1, p. 49–73, jan. 2009.
- 156 BRATTHALL, L.; JØRGENSEN, M. Can you trust a single data source exploratory software engineering case study? *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 7, p. 9–26, March 2002. ISSN 1382-3256.
- 157 APEL, S.; KÄSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, v. 8, p. 49–84, 07 2009.

This volume has been typeset in L^AT_EX with the UFBAThesis class (<http://www.dcc.ufba.br/~flach/ufbathesis>). For details about this document, click [here](#).