FEDERAL UNIVERSITY OF CEARÁ

SCIENCE CENTER

COMPUTING DEPARTMENT

GRADUATE PROGRAM IN COMPUTER SCIENCE

LUAN PEREIRA LIMA

UNDERSTANDING THE EFFECTIVENESS OF EXCEPTION HANDLING TESTING IN LONG-LIVED JAVA LIBRARIES

FORTALEZA

2019

LUAN PEREIRA LIMA

UNDERSTANDING THE EFFECTIVENESS OF EXCEPTION HANDLING TESTING IN LONG-LIVED JAVA LIBRARIES

Dissertation presented to the Graduate Program in Computer Science of the Science Center of the Federal University of Ceará, as a partial requirement to obtain the title of Master in Computer Science. Concentration Area: Software Engineering.

Advisor: Prof. Dr. Lincoln S. Rocha

Co-supervisor: Profa. Dra. Carla I. Moreira Bezerra

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

L698u Lima, Luan Pereira.
Understanding the effectiveness of exception handling testing in long-lived java libraries / Luan Pereira Lima. – 2019.
60 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2019.
Orientação: Prof. Dr. Lincoln Souza Rocha.
Coorientação: Profa. Dra. Carla Ilane Moreira.

1. Exception Handling Testing. 2. Mutation Analysis. 3. Adequacy Measurement. 4. Effectiveness Measurement. 5. Exploratory Study. I. Título.

CDD 005

LUAN PEREIRA LIMA

UNDERSTANDING THE EFFECTIVENESS OF EXCEPTION HANDLING TESTING IN LONG-LIVED JAVA LIBRARIES

Dissertation presented to the Graduate Program in Computer Science of the Science Center of the Federal University of Ceará, as a partial requirement to obtain the title of Master in Computer Science. Concentration Area: Software Engineering.

Approved on:

EXAMINATION COMMITTEE

Prof. Dr. Lincoln S. Rocha (Advisor)
Universidade Federal do Ceará (UFC)

Profa. Dra. Carla I. Moreira Bezerra (Co-supervisor)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Henrique Mendes Maia
Universidade Estadual do Ceará(UECE)

Prof. Dr. João Bosco Ferreira Filho
Universidade Federal do Ceará (UFC)

Prof. Dr. Matheus Henrique Esteves Paixão
Universidade de Fortaleza (UNIFOR)

My family, friends and companion, for the great support in this journey, which is to obtain the title of master.

ACKNOWLEDGEMENTS

To Prof. Dr. Lincoln Rocha for guiding me, more than just in my Master's Dissertation, but in the entire journey itself.

Profa. Dr. Carla Ilane for guiding me and making it possible to complete this master's degree.

To Prof. Dr. Matheus Paixao for helping to write and submission of the scientific article developed from this study.

“Persistence is the path to success.”

(Charles Chaplin)

RESUMO

Linguagens de programação modernas (por exemplo, Java e C#) fornecem recursos para separar o código de tratamento de erros do código regular, buscando melhorar a compreensão e manutenção do software. No entanto, a forma como o código Exception Handling (EH) é estruturado em tais linguagens pode levar a múltiplos fluxos de controle, diferentes e complexos, o que pode afetar a testabilidade do software. Estudos anteriores relataram que o código EH é normalmente negligenciado, não é bem testado e seu uso indevido pode levar à degradação da confiabilidade e falhas catastróficas. No entanto, pouco se sabe sobre a relação entre as práticas de teste e a eficácia do teste EH. Neste estudo exploratório, (i) medimos o grau de adequação do teste EH em relação aos critérios de cobertura de código (instrução, ramificação e método); e (ii) avaliou a eficácia do teste de EH medindo sua capacidade de detectar falhas injetadas artificialmente (isto é, mutantes) usando 7 operadores de mutação de EH. Nosso estudo foi realizado usando suítes de teste de 27 bibliotecas Java de vida longa de sistemas de código aberto. Nossos resultados mostram que as instruções e ramificações dentro de blocos de captura e instruções de lançamento são menos cobertas, com significância estatística, do que as instruções e ramificações gerais. Apesar disso, a maioria dos sistemas estudados apresentou suítes de teste capazes de detectar mais de 70% das falhas injetadas.

Palavras-chave: Teste de tratamento de exceções. Análise de mutação. Medição de adequação. Medição de eficácia. Estudo exploratório.

ABSTRACT

Modern programming languages (e.g., Java and C#) provide features to separate error-handling code from regular code, seeking to enhance software comprehensibility and maintainability. Nevertheless, the way EH code is structured in such languages may lead to multiple, different, and complex control flows, which may affect the software testability. Previous studies have reported that EH code is typically neglected, not well tested, and its misuse can lead to reliability degradation and catastrophic failures. However, little is known about the relationship between testing practices and EH testing effectiveness. In this exploratory study, we (i) measured the adequacy degree of EH testing concerning code coverage (instruction, branch, and method) criteria; and (ii) evaluated the effectiveness of the EH testing by measuring its capability to detect artificially injected faults (i.e., mutants) using 7 EH mutation operators. Our study was performed using test suites of 27 long-lived Java libraries from open source systems. Our results show that instructions and branches within `catch` blocks and `throw` instructions are less covered, with statistical significance, than the overall instructions and branches. Nevertheless, most of the studied systems presented test suites capable of detecting more than 70% of the injected faults.

Keywords: Exception Handling Testing. Mutation Analysis. Adequacy Measurement. Effectiveness Measurement. Exploratory Study

LIST OF FIGURES

Figure 1 – Internal steps performed by XaviEH when evaluating the EH testing practices of a software system. 30
Figure 2 – Overall code coverage boxplots of the 27 studied libraries (Instruction Coverage (IC), Branch Coverage (BC), and Covered Method (MC)). 34
Figure 3 – XaviEH architecture, most important classes for operation. 34
Figure 4 – Distribution of general EH-related code coverage metrics for the systems under study. 38
Figure 5 – EH instruction code coverage boxplots of studied libraries. 38
Figure 6 – EH branch code coverage boxplots of studied libraries. 39
Figure 7 – Overall EH and non-EH code coverage boxplots of studied libraries. 43
Figure 8 – The EH and non-EH instruction coverage boxplots of studied libraries. . . . 44
Figure 9 – The EH and non-EH branch coverage boxplots of studied libraries. 45
Figure 10 – Mutation scores distribution bloxpots. 47

LIST OF TABLES

Table 1 – Summary of selected libraries. While the first chapter depicts the libraries from Apache Commons, the second chapter indicates the selected libraries from other ecosystems. We provide the version we studied of each library followed by size metrics, such as Lines of code (LOC), number of `throw` instructions, number of `try` blocks etc. 29

Table 2 – Mutation operators employed in this study. All operators are based on real-world defects from open-source systems. 32

Table 3 – Code coverage metrics computed by XaviEH. The first set of metrics are computed considering the system's entire code base. The second set of coverage metrics are specific for exception handling code. Finally, the third set is composed of complements to the exception handling coverage metrics. . . . 33

Table 4 – Computed general code coverage measures per library. 37

Table 5 – Computed complementary code coverage measures per library. 41

Table 6 – Summary of hypothesis statement and the statistics test results. 42

Table 7 – Details of the mutation testing analysis for each library. Column `L` indicates the number of live mutants, column `D` indicates the number of killed mutants, and `S` indicates the mutation score. 46

Table 8 – The ranks and average rank of mutation scores. 49

LIST OF ABBREVIATIONS AND ACRONYMS

EH	Exception Handling
IC	Instruction Coverage
BC	Branch Coverage
MC	Covered Method
LOC	Lines of code
CBR	Catch Block Replacement
CBI	Catch Block Insertion
CBD	Catch Block Deletion
PTL	Placing Try Block Later
CRE	Catch and Rethrow Exception
CCD	Catch Clauses Deletion
TSD	Throw Statement Deletion
ENC	Exception Name Change
FCD	Finally Clause Deletion
EHM	Exception Handling Modification
FBD	Finally Block Deletion
EH_IC	Exception Handling Instruction Coverage
EH_BC	Exception Handling Branch Coverage
TRY_IC	Try Instruction Coverage
TRY_BC	Try Branch Coverage
CATCH_IC	Catch Instruction Coverage
CATCH_BC	Catch Branch Coverage
FINALLY_IC	Finally Instruction Coverage
FINALLY_BC	Finally Branch Coverage
THROW_IC	Throw Instruction Coverage
THROWS_MC	Throws Method Coverage
NON_EH_IC	Non-Exception Handling Instruction Coverage
NON_EH_BC	Non-Exception Handling Branch Coverage
NON_TRY_IC	Non-Try Instruction Coverage
NON_TRY_BC	Non-Try Branch Coverage
NON_CATCH_IC	Non-Catch Instruction Coverage

NON_CATCH_BC	Non-Catch Branch Coverage
NON_FINALLY_IC	Non-Finally Instruction Coverage
NON_FINALLY_BC	Non-Finally Branch Coverage
NON_THROW_IC	Non-Throw Instruction Coverage
NON_THROWS_MC	Non-Throws Method Coverage
XML	Extensible Markup Language
CSV	Comma Separated Values
KS	Kolmogorov–Smirnov test
MW	Mann-Whitney test
CD	Critical Difference

LIST OF SYMBOLS

$\downarrow$	Min
$\uparrow$	Max
$\bar{x}$	Mean
σ	Standard Deviation
q_1	First Quartile
q_3	Third Quartile

CONTENTS

1	**INTRODUCTION**	16
2	**BACKGROUND**	19
2.1	**Software Test Criteria and Adequacy**	19
2.2	**Mutation Testing and Analysis**	19
2.3	**Java Exception Handling**	20
3	**RELATED WORK**	22
3.1	**Exception Handling and Software Bugs**	22
3.2	**Exception Handling Testing**	24
3.3	**Code Coverage and Defect-Free Software**	25
4	**EXPERIMENTAL DESIGN**	27
4.1	**Selection of Long-lived Java Libraries**	28
4.2	**Assessing Exception Handling Testing with XaviEH**	30
4.3	**Mutation Operators and Analysis**	31
4.4	**Code Coverage Metrics**	32
4.5	**Preliminary Observation of the Libraries' Overall Coverage**	33
4.6	**XaviEH architecture**	34
5	**STUDY RESULTS**	36
5.1	**RQ1. What is the test coverage of EH code in long-lived Java libraries?**	36
5.2	**RQ2. What is the difference between EH and non-EH code coverage in long-lived Java libraries?**	40
5.3	**RQ3. What is the effectiveness of EH testing in long-lived Java libraries?**	45
5.4	**RQ4. To what extent are there EH bugs that are statistically harder to detect by test suites of long-lived Java libraries?**	48
6	**DISCUSSION**	51
6.1	**On the Adequacy of EH Testing**	51
6.2	**On the Effectiveness of EH Testing**	51
6.3	**On the Usefulness of XaviEH**	52
7	**THREATS TO THE VALIDITY**	53
7.1	**Conclusion Validity**	53
7.2	**Internal Validity**	53
7.3	**Construct Validity**	53

7.4 **External Validity** 54
8 **CONCLUSION AND FINAL REMARKS** 55
BIBLIOGRAPHY 56

1 INTRODUCTION

EH is a forward-error recovery technique used to improve software robustness (SHAHROKNI; FELDT, 2013). An exception models an abnormal situation - detected at run time - that disrupts the normal control flow of a program (GARCIA *et al.*, 2001). When this happens, the EH mechanism deviates the normal control flow to the abnormal (exceptional) control flow to deal with such situation. Mainstream programming languages (e.g., Java, Python, and C#) provide built-in facilities to structure the exceptional control flow using proper constructs to specify, in the source code, where exceptions can be raised, propagated, and properly handled (CACHO *et al.*, 2014a).

Recent studies have investigated the relationship between EH code and software maintainability (CACHO *et al.*, 2014b), evolvability (OSMAN *et al.*, 2017), architectural erosion (FILHO *et al.*, 2017), robustness (CACHO *et al.*, 2014a), bug appearance (EBERT *et al.*, 2015), and defect-proneness (SAWADPONG; ALLEN, 2016). Such studies have shown that the quality of EH code is directly linked to the overall software quality (PáDUA; SHANG, 2017b; PáDUA; SHANG, 2018). To ensure and assess the quality of EH code, developers make use of software testing, which, in this context, is referred to as EH testing (SINHA; HARROLD, 2000; Martins *et al.*, 2014; ZHANG; ELBAUM, 2014).

Despite the importance and the existence of usage patterns and guidelines for EH testing (WIRFS-BROCK, 2006; BLOCH, 2008; GALLARDO *et al.*, 2014), this is a commonly neglected activity by developers (mostly by novice ones) (SHAH *et al.*, 2010). Moreover, EH code is claimed as the least understood, documented, and tested part of a software system (SHAH *et al.*, 2008; SHAH; HARROLD, 2009; SHAH *et al.*, 2010; RASHKOVITS; LAVY, 2012; KECHAGIA; SPINELLIS, 2014; CHANG; CHOI, 2016). In addition, (EBERT *et al.*, 2015) have found in a survey with developers that about 70% of the software companies do not test and have no specific testing technique for EH code. This is a worrisome finding given the importance of EH testing for ensuring software quality.

In the current landscape of software development, researchers commonly study open-source systems to acquire insights on many aspects of software development and quality, including architectural practices (PAIXAO *et al.*, 2017), refactoring (BAVOTA *et al.*, 2015), evolution (KOCH, 2007) and bug fixing (VIEIRA *et al.*, 2019), to mention a few. However, to the best of our knowledge, there is no empirical study that observes and evaluates EH testing practices in open-source software. As a result, the software engineering community lacks a thorough

and concise understanding of good and openly available EH testing practices. This prevents the further creation of EH testing guidelines that are based on real-world software and practices instead of textbooks and rules of thumb, such as the ones currently available (WIRFS-BROCK, 2006; BLOCH, 2008; GALLARDO *et al.*, 2014).

Nevertheless, to evaluate the quality of software testing as a whole, and EH testing in specific, is not a trivial task. First, one needs to define what constitutes a good test. Early in 1975, (GOODENOUGH; GERHART, 1975) defined the concept of test criterion as a way to precisely state what constitutes a suitable software test. Currently, the code coverage (e.g., instruction, branch, and method) criteria have been widely used as a proxy for testing quality (IVANKOVIć *et al.*, 2019; YANG *et al.*, 2019). However, recent studies provide evidence that high test coverage alone is not sufficient to avoid software bugs (Antinyan *et al.*, 2018; Kochhar *et al.*, 2017). In parallel, mutation testing (a.k.a, mutation analysis) provides a way to evaluate the effectiveness of test suites by artificially injecting bugs that are similar to real defects (PAPADAKIS *et al.*, 2018). Recent studies have shown that mutant detection is significantly correlated with real fault detection (JUST *et al.*, 2014).

Hence, in this study, we report on the first empirical study that assesses and evaluates the practices of EH testing in open-source software systems. We developed a tool, called XaviEH, to assist in these analyses. XaviEH employs both coverage and mutation analysis as proxies for the quality of EH testing in a certain system. In addition, XaviEH uses tailored quality criteria for EH code, including EH-specific coverage measures and mutation operators. In total, XaviEH measured the adequacy and effectiveness of EH testing of 27 long-lived Java libraries. Finally, based on the analysis by XaviEH, we ranked these libraries to assess which ones present significantly better indicators of EH testing quality.

The main contributions of this master dissertation are listed as follows:

- The first empirical study to evaluate EH testing practices in open-source software.
- A tool, called XaviEH, to automatically assess the adequacy and effectiveness of EH testing in a software system.
- A dataset concerning the analysis of 27 long-lived Java libraries regarding their EH testing practices (LIMA *et al.*, 2020).

Overall, our findings suggest that EH code is, in general, less covered than regular code (i.e. non-EH). Additionally, we provide evidence that the code within the `catch` blocks and `throw` statements have a low coverage degree. However, despite not being well-covered, the

mutation analysis shows that the test suites are able to detect artificial EH-related faults.

The remainder of this master dissertation is organized as follows. Chapter 2 provides a background for our study. Chapter 3 addresses the related work. Chapter 4 presents the experimental design of our study. The study results are presented in Chapter 5. In Chapter 6, our results and implications for researchers and practitioners are discussed. Chapter 7 presents the threats to validity, and at last, Chapter 8 concludes the master dissertation and points out directions for future work.

2 BACKGROUND

In this Chapter, we describe the general concepts and definitions that provide a background to our study.

2.1 Software Test Criteria and Adequacy

(GOODENOUGH; GERHART, 1975) state that a software test adequacy criterion defines "*what properties of a program must be exercised to constitute a 'thorough' test, i.e., one whose successful execution implies no errors in a tested program*". To guarantee the correctness of adequately tested programs, they proposed reliability and validity requirements of test criteria (ZHU *et al.*, 1997). The former requires that a test criterion always produce consistent test results (i.e., if the program is tested successfully on a certain test set that satisfies the criterion, then the program should also be tested successfully on all other test sets that satisfies the criterion). The later requires that the test should always produce a meaningful result concerning the program under testing (i.e., for every error in a program, there exists a test set that satisfies the criterion and it is capable of revealing the error).

Code coverage (also known as test coverage) is a metric to assess the percentage of the source code executed by a test suite. Code coverage is commonly employed as a proxy for test adequacy (Kochhar *et al.*, 2017). The percentage of code executed by test cases can be measured according to various criteria, such as (Antinyan *et al.*, 2018): statement coverage, branch coverage, and function/method coverage. Statement coverage is the percentage of statements in a source file that have been exercised during a test run. Branch coverage is the percentage of decision blocks in a source file that have been exercised during a test run. Function/Method coverage, is the percentage of all functions/methods in a source file that have been exercised during a test run. For the rest of this master dissertation, we use code coverage and test coverage interchangeably.

2.2 Mutation Testing and Analysis

Mutation analysis is a procedure for evaluating the degree to which a program is properly tested, that is, to measure a test suite's effectiveness. Mutation testing evaluates a certain test suite by injecting artificial defects in the source code. In this context, a test suite that is able to identify artificial defects is likely to be able to pinpoint real defects when these occur. Hence,

to maximize mutation testing's ability to measure the effectiveness of a test suite, one must inject artificial defects that are as close as possible to real defects (JUST *et al.*, 2014; PAPADAKIS *et al.*, 2018).

A version of a software system with an artificially inserted fault is called a mutant. Mutation operators are rule-based program transformations used to create mutants from the original source code of a software system. When executing the test suite of a system in both the original and mutant code, if the mutant and the original code produce different outputs in at least one test case, the fault is detected, i.e, the mutant can be killed by the test suite. Consider $M(s)$ to be the set of mutants created for system s and $KM(s)$ the set of killed mutants for system s. Mutation score, as detailed in Equation 2.1, indicates the percentage of dead mutants compared to the mutants that are not equivalent to the original program (ZHU *et al.*, 1997). Mutation score indicates the effectiveness of a certain test suite, as it evaluates the test suite's ability to find defects.

$$MutationScore(s) = \frac{|M(s)|}{|KM(s)|} \times 100 \tag{2.1}$$

A certain variant of a software system is considered a first-order mutant when only a single artificial defect has been introduced. Differently, higher-order mutants are the ones generated by combining more than one mutation operator. We focus on first-order mutants for this study (see Section 4.2).

2.3 Java Exception Handling

In the Java programming language, "*an exception is an event, which occurs during the execution of a program, which disrupts the normal flow of the program's instructions*" (GALLARDO *et al.*, 2014). When an error occurs inside a method, an exception is raised. In Java, the raising of an exception is called *throwing*. Exceptions are represented as objects following a class hierarchy and can be divided into two categories: checked and unchecked. Checked exceptions are all exceptions that inherits, directly or indirectly, from Java's `Exception` class, except those ones that inherits, directly or indirectly, from `Error` or `RuntimeException` classes, named unchecked ones. Checked exceptions represent exceptional conditions that, hypothetically, a robust software should recover from. Unchecked exceptions represent an internal (`RuntimeException`) or an external (`Error`) exceptional conditions that a software usually

cannot anticipate or recover from. In Java, only the handling of checked exceptions is mandatory, which obligate developers to write error-handling code to catch and handle them.

When an exception is raised, the execution flow is interrupted and deviated to a specific point where the exceptional condition is handled. In Java, exceptions can be raised using the `throw` statement, signaled using the `throws` statement, and handled in the `try-catch-finally` blocks. The "`throw new E()`" statement is an example of *throwing* the exception `E`. The "`public void m() throws E,T`" shows how the `throws` clause is used in the method declaration to indicate the signaling of exceptions `E` and `T` to the method that call `m()`.

The `try` block is used to enclose the method calls that might throw an exception. If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. Handlers are represented by `catch` blocks that are written right below the respective `try` block. Multiple `catch` blocks can be associated with a `try` block. Each `catch` block catches a specific exception type and encloses the exception handler code. The `finally` block is optional, but when declared, it always executes when the `try` block finishes, even with or without an exception occurring and/or being handled. Finally blocks are commonly use for coding cleanup actions.

3 RELATED WORK

In this chapter, we present the related work that, in some way, are related to our study.

3.1 Exception Handling and Software Bugs

Previous work has investigated and provided evidence on the positive correlation between exception handling code and software defect proneness (MARINESCU, 2011; MARINESCU, 2013). In fact, this correlation emerges from sub-optimal exception handling practices (i.e., anti-patterns and flow characteristics) current adopted by software developers (SAWADPONG; ALLEN, 2016; PáDUA; SHANG, 2018). Additionally, the exception handling is usually neglected by developers (mainly by novices ones) and is considered as one of the least understood, documented, and tested part of a software system (SHAH *et al.*, 2010; ZHANG; ELBAUM, 2014; CHANG; CHOI, 2016; OLIVEIRA *et al.*, 2018).

The studies conducted by (BARBOSA *et al.*, 2014) and (EBERT *et al.*, 2015) gather evidence that erroneous or improper usage of exception handling can lead to a series of fault patterns, named "exception handling bugs". This kind of faults refer to bugs in which the primary source is related to (i) the exception definition, throwing, propagation, handling or documentation; (ii) the implementation of cleanup actions; and (iii) wrong throwing or handling (i.e., when the exception should be thrown or handled and it is not). (BARBOSA *et al.*, 2014) categorizes 10 causes of exception handling bugs, analyzing two open source projects, Hadoop and Apache Tomcat. (EBERT *et al.*, 2015) extends (BARBOSA *et al.*, 2014) study, presenting a comprehensive classification of exception handling bugs based on a survey of 154 developers and the analysis of 220 exception handling errors reported from two open source projects, Apache Tomcat and Eclipse IDE. (KECHAGIA; SPINELLIS, 2014) studied undocumented runtime exceptions thrown by the Android platform and third-party libraries. They mined 4,900 different stack traces from 1,800 apps looking for undocumented API methods with undocumented exceptions participating in the crashes. They found that 10% of crashes might have been avoided if the correspondent runtime exceptions had been properly documented.

(PáDUA; SHANG, 2017b; PáDUA; SHANG, 2017a; PáDUA; SHANG, 2018) conducted a series of studies concerning exception handling and software quality. In the first study, they conducted an investigation on the prevalence of exception handling anti-patterns

across 16 open source projects (Java and C#). They claim that the misuse of exception handling can cause catastrophic software failures, including application crashes. They found that all 19 exception handling anti-patterns taken into account in the study are broadly present in all subject projects, however, only 5 of them (unhandled exception, generic catch, unreachable handler, over-catch, and destructive wrapping) are prevalent. Next, (PáDUA; SHANG, 2017a) conducted a study revisiting the exception handling practices by analyzing the flow of exceptions from the source of exceptions until its handling blocks in 16 open-source projects (Java and C#). Once researchers understood that exception handling practices may lead to software failures, their identification highlight the opportunities of leveraging automated software analysis to assist in exception handling practices.

Finally, the third study of (PáDUA; SHANG, 2018) focuses on understanding the relationship between exception handling practices and post-release defects. They investigated the relationship between post-release defect proneness and: (i) exception flow characteristics; and (ii) 17 exception handling anti-patterns. Their finds suggests that development teams should find a way to improve their exception handling practices and avoid the anti-patterns (e.g., dummy handler, generic catch, ignoring interrupted exception, and log and throw) that are found to have a relationship with post-release defects.

(COELHO *et al.*, 2017) mined 6,000 stack traces from over 600 open source projects issues on GitHub and Google Code searching for bug hazards regarding exception handling. Additionally, they surveyed 71 developers involved in at least one of the projects analyzed. As a result, they found four bug hazards that may cause bugs in Android applications: (i) cross-type exception wrapping; (ii) undocumented unchecked exceptions raised by the Android platform and third-party libraries; (iii) undocumented check exceptions signaled by native C code; and (iv) programming mistakes made by developers. The survey's results corroborate the stack trace findings, indicating that developers are unaware of frequently occurring undocumented exception handling behavior.

Similar to the studies cited, our study investigates practices for EH testing in 27 long-lived Java libraries from open-source systems that represent systems with more than 11 years of active development. We generated a total of $12,331$ software mutants and the systems present effective test suites for EH code, where more than 70% of the defects being identified.

3.2 Exception Handling Testing

(JI *et al.*, 2009) proposes 5 types of exception handling code mutants: Catch Block Replacement (CBR), Catch Block Insertion (CBI), Catch Block Deletion (CBD), Placing Try Block Later (PTL), Catch and Rethrow Exception (CRE). (KUMAR *et al.*, 2011) develops 5 types of mutants for exception handling code, namely: Catch Clauses Deletion (CCD), Throw Statement Deletion (TSD), Exception Name Change (ENC), Finally Clause Deletion (FCD) and Exception Handling Modification (EHM). These operators try to replace, insert, delete some capture blocks, add statements to re-launch a capture block, and try to rearrange try blocks by including statements with some relevant references after the capture blocks. In our study, we employed 7 mutation operators: 5 mutation operators (CBR, CBI, CBD, PTL, and CRE); and, 2 operators Finally Block Deletion (FBD) and TSD. Our study generated a total of $12,331$ software mutants as follows: 98 (CBI), $2,519$ (CBD), $2,519$ (CRE), 404 (FBD), 84 (PTL), 80 (CBR), and $6,627$ (TSD).

The work of (ZHANG; ELBAUM, 2014) presents an automated approach to support the detection of faults in exception handling code that deals with external resources. The study revealed that 22% of the confirmed and fixed bugs have to do with poor exceptional handling code, and half of those correspond to interactions with external resources. In our study we identified as a result that despite presenting high coverage for instruction and branches in the overall source code and EH code, the tests are still mostly exercising non-exceptional flows within the programs, where the exception behaviors are not being tested.

In a recent study, (ZHAI *et al.*, 2019) undertook a study of code coverage in popular Python projects: flask, matplotlib, pandas, scikit-learn and scrapy. In this study, the authors found that coverage depends on control flow structure, with more deeply nested statements being significantly less likely to be covered. Other findings of the study were that the age of a line per se has a small (but statistically significant) positive effect on coverage. Finally, they found that the kind of statement (e.g., try, if, except, and raise) has varying effects on coverage, with exception handling statements being covered much less often. The results suggest that developers in Python projects have difficulty writing test sets that cover deeply-nested and error-handling statements, and might need assistance covering such code. Our study used long-lived Java libraries, and a deduction from the identified results was that despite presenting high coverage for instruction and branches in the overall source code and EH code, the tests are still mostly exercising non-exceptional flows within the programs, where the exception behaviors are not

being tested.

3.3 Code Coverage and Defect-Free Software

(INOZEMTSEVA; HOLMES, 2014a) conducted one of the first large studies that investigated the correlation between code coverage and test effectiveness. Their study took into account 31,000 test suites generated for five large Java systems. They measured code coverage (statement, branch, and modified condition) using these test suites and employed mutation testing to evaluate the effectiveness of such test suites in revealing the injected faults. They found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for this purpose. (KOCHHAR *et al.*, 2015) conducted a study seeking out to investigate the correlation between code coverage and its effectiveness in real bugs. The experiment was performed taking into account 67 and 92 real bugs from Apache HTTPClient and Mozilla Rhino, respectively. They used a tool, called Randoop, to generate random test suites with varying levels of coverage and run them to analyze the capability of these synthetic test suites in detecting the existing bugs in both systems. They found that there is a statistically significant correlation between code coverage and bug detection effectiveness.

(Kochhar *et al.*, 2017) performed a large scale study concerning the correlation between real bugs and code coverage of exiting test suits. This study took into account 100 large open-source Java projects. They extracted real bugs recorded in the project's issue tracking system after the software release and analyzed the correlations between code coverage and these bugs. They found that the coverage of actual test suites has an insignificant correlation with the number of bugs that are found after the software release.

(SCHWARTZ *et al.*, 2018) argue that previous work provide mix results concerning the correlation between code coverage and test effectiveness (i.e., some studies provide evidence on a statistically significant correlation between these two factors, while others do not). Thus, they hypothesize that the fault type is one of the sources that may be leading to these mixed results. To investigate this hypothesis, they have studied 45 different types of faults and evaluated how effectively human-created test suites with high coverage percentages were able to detect each type of fault. The study was performed on 5 open-source projects (Commons Compress, Joda Time, Commons Lang, Commons Math, and JSQL Parser), which have at least 80% statement coverage. The mutation testing technique was employed to seed 45 types of faults in the program's code in order to evaluate the effectiveness of the existing unit test suites in the

detection of such fault types. Their findings showed that, with statistical significance, there were specific types of faults found less frequently than others. Additionally, based on their findings, they suggest developers should put more focus on improving test oracles strength along with code coverage to achieve higher levels of test effectiveness.

Our study analyzes the test coverage compared to EH code. We developed a tool, called XaviEH, which employs both coverage and mutation analysis as proxies for the quality of EH testing in a certain system. Our findings suggest that EH code is, in general, less covered than regular code (i.e. non-EH).

4 EXPERIMENTAL DESIGN

Our study aims at investigating practices for EH testing in open-source systems. To achieve this, we selected 27 long-lived Java libraries to serve as subjects in our empirical evaluation (see Section 4.1). Hence, we ask the following research questions:

RQ1. *What is the test coverage of EH code in long-lived Java libraries*?

First, we measure EH testing adequacy in terms of code coverage measures. We employ a variant of long-established coverage criteria in the literature (see Chapter 4.4) to provide the first insight regarding the extent to which the test suites of the studied Java libraries exercise EH code.

RQ2. *What is the difference between EH and non-EH code coverage in long-lived Java libraries*?

In addition to measuring the coverage of EH code, we also measure the coverage of non-EH code in each library. By controlling the EH coverage with its non-EH counterpart, we can reason on how EH testing differs from other testing activities to better understand how (in)adequate EH testing may be.

RQ3. *What is the effectiveness of EH testing in long-lived Java libraries*?

We employ mutation testing to assess the effectiveness of EH testing. By leveraging EH mutation operators derived from real EH bugs, we create artificial defects that are similar to EH bugs found in real-world software systems. Next, we measure the mutation score and use it as a proxy for the effectiveness of EH testing.

RQ4. *To what extent are there EH bugs that are statistically harder to detect by test suites of long-lived Java libraries?*

We employ a combination of the Friedman (FRIEDMAN, 1940) and Nemenyi (DEMsAR, 2006) tests to statistically assess whether there are types of EH bugs that are more difficult to detect by the studied libraries' test suites than others. We aspire to find a set of EH bugs that developers must be aware of during testing, aiming at fostering knowledge and improving the effectiveness of EH testing.

The rest of this chapter details the methodology employed in our empirical study to answer the research questions presented above. The complete dataset, source code and results for this empirical study is available at our replication package (LIMA *et al.*, 2020).

4.1 Selection of Long-lived Java Libraries

Our study focuses on the study of EH testing. However, EH is not a trivial activity in software development (SHAH *et al.*, 2010). First, the need for EH commonly arises as systems evolve and are exposed to a wide range of usage scenarios that expose runtime flaws (CACHO *et al.*, 2014b; PáDUA; SHANG, 2018; CHEN *et al.*, 2019). Second, EH testing is considered more challenging than non-EH testing due to its complex runtime and 'flakiness' nature (ZHANG; ELBAUM, 2014; ECK *et al.*, 2019). Thus, to properly study EH testing, we need long-lived subject systems that attend to a large number of users and usage scenarios. In addition, we need systems with reputably good quality to maximize the chances that the development team is versed and employ good practices in both EH handling and testing.

Therefore, we turned our attention to the Apache Software Foundation ecosystem[1]. The Apache Foundation is a well-known open-source software community that leads the continuous development of open-source general-purpose software solutions. Not only this community hosts long-lived systems in active development (Apache's Commons Collections library, for instance, is now 17 year old) but it is also known to follow good software engineering practices, where its systems have been the object of a plethora of previous empirical studies (SHI *et al.*, 2011; BARBOSA *et al.*, 2014; AHMED *et al.*, 2016; SCHWARTZ *et al.*, 2018; HILTON *et al.*, 2018; Digkas *et al.*, 2018; VIEIRA *et al.*, 2019; Zhong; Mei, 2019).

For this particular study, we considered libraries of the Apache Commons Project, which is an Apache project focused on all aspects of reusable Java components[2]. We focused on libraries because they tend to be more generic and present more usage scenarios than other systems. As a selection criteria for our study, a library should: (i) be developed in Java; (ii) employ Maven or Graddle as build system; (iii) present an automatically executable and passing test suite; (iv) be a long-lived system; and (v) be correctly handled by Spoon (PAWLAK *et al.*, 2016), one of the the tools we used to build XaviEH (see Chapter 4.3). To identify long-lived libraries, we computed the distribution of all libraries' age in years. Hence, we considered long-lived systems, all libraries above the 3rd quartile in the distribution, which, for this study, represent systems with more than 11 years of active development.

As a result, we selected 21 libraries out of the 96 available in Apache Commons. We provide details about each selected library in the first chapter of Table 1. Nevertheless, while fit

[1] https://apache.org/index.html#projects-list
[2] https://commons.apache.org/

Table 1 – Summary of selected libraries. While the first chapter depicts the libraries from Apache Commons, the second chapter indicates the selected libraries from other ecosystems. We provide the version we studied of each library followed by size metrics, such as LOC, number of `throw` instructions, number of `try` blocks etc.

Library	Version	#LoC	#Classes	#Throw	#Try	#Catch	#Finally	#Years
BCEL	6.2	61100	344	406	147	143	5	18
BeanUtils	1.9.3	32150	98	364	126	164	0	18
CLI	1.4	6245	21	29	12	11	1	17
Codec	1.11	18559	55	97	28	22	8	16
Collections	4.2	68319	270	725	28	44	1	18
Compress	1.18	47741	183	425	137	73	39	16
Configuration	2.4	66869	178	306	235	159	96	16
DBCP	2.5	23132	50	279	796	846	23	18
DbUtils	1.7	8850	39	46	41	40	20	16
Digester	3.3.2	22858	132	110	68	72	7	18
Email	1.5	6115	19	74	36	32	9	15
Exec	1.3	4600	26	29	23	23	6	14
FileUpload	1.3.3	6884	23	50	25	26	6	17
Functor	1.0	17617	135	115	0	0	0	16
IO	2.6	28691	112	292	106	80	8	17
Lang	3.8.1	78174	124	380	76	81	5	17
Math	3.6.1	223110	740	1494	118	124	4	16
Net	3.6	47107	175	159	174	180	24	17
Pool	2.6.1	13629	33	79	132	70	79	18
Proxy	1.0	4112	37	36	23	31	0	11
Validator	1.6	17677	62	68	40	49	1	17
Gson	2.8.5	14863	52	222	56	75	5	11
Hamcrest	2.1	7834	77	19	12	13	0	13
Jsoup	1.11.3	18111	55	46	33	32	3	11
JUnit	4.12	17200	149	101	99	119	17	19
Mockito	2.23.11	33505	297	236	91	98	20	12
X-Stream	1.4.11.1	37475	313	461	248	362	19	16

for our empirical study, to consider only libraries from the Apache community would represent a threat to the study's generability and diversity (NAGAPPAN *et al.*, 2013). Hence, we selected 6 additional non-Apache libraries that adhere to the same inclusion criteria discussed above. These were selected considering their ranking on open source platforms, such as GitHub, and personal experience from the authors in using these libraries. The additional libraries are depicted in the second chapter of Table 1. In total, our empirical study considered 27 long-lived libraries from different open source ecosystems.

After selecting the 27 libraries employed in the study, we performed the data collection. On March 2019, we downloaded the latest available release of each library in which we could automatically build and execute the test suite without any failing test.

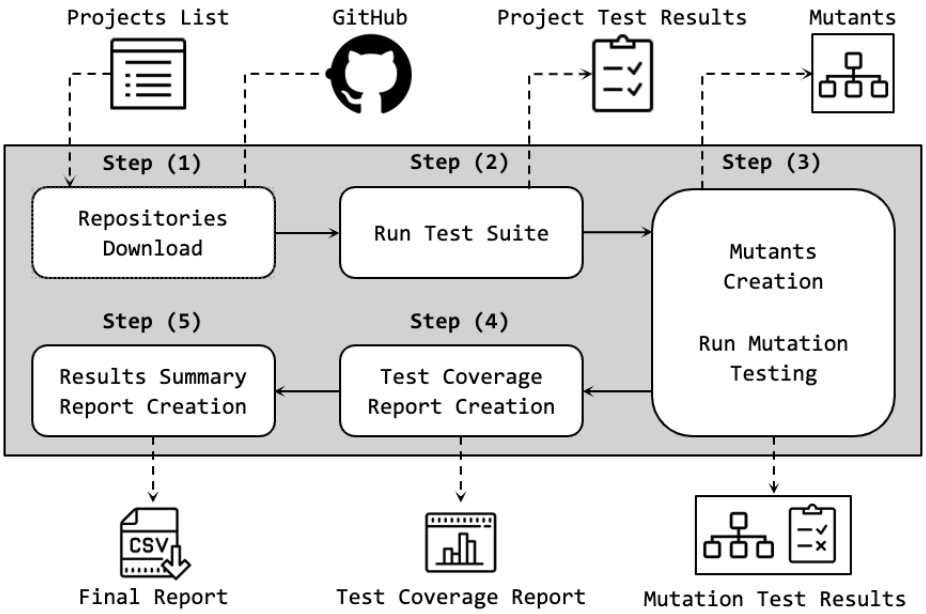

Figure 1 – Internal steps performed by XaviEH when evaluating the EH testing practices of a software system.

4.2 Assessing Exception Handling Testing with XaviEH

To perform our study, we developed the XaviEH tool. Given a certain software system, XaviEH is able to automatically perform an analysis regarding the the system's practices on EH testing. It provides a report on the adequacy and effectiveness of the system's test suite when testing EH code. Figure 1 illustrates the execution steps of XaviEH for a software system.

In Step (1), XaviEH obtains the system's source code. In case the system is hosted on GitHub, one can provide the GitHub URL, and XaviEH will use JGit[3] to download the source code. Otherwise, one can simply provide the source code to XaviEH.

In Step (2), XaviEH executes the system's test suite to verify that all tests are passing. This is necessary to ensure that the next steps will be correctly executed. XaviEH uses the `maven-invoker`[4] and `gradle-tooling-api`[5] libraries to run the tests automatically.

Step (3) involves the mutation analysis. XaviEH generates all possible first-order mutants of the system being analyzed. To do this, XaviEH first searchs the system's source code to identify all classes eligible for mutation. In this study, a class is considered eligible if it has any code structure that can be affected (mutated) by at least one of the seven mutation operators we employ (see Chapter 4.3). By doing so, XaviEH creates an in-memory data structure that tracks eligible classes and mutation operators that can be applied to each class.

[3] https://www.eclipse.org/jgit/
[4] https://maven.apache.org/shared/maven-invoker/
[5] https://docs.gradle.org/current/userguide/embedding.html

Next, for each eligible class, XaviEH applies all mutation operators that can be applied to the class, recording which operator was applied to which class. A mutation operator may be applied to an eligible class more than once. In this case, XaviEH ensures that successive changes made by a specific mutation operator within the same eligible class do not affect the same location twice, preventing duplicate mutants from being generated. To perform both code search and mutant generation tasks, XaviEH uses Spoon[6], a library for parsing and transforming Java source code.

For each mutant, XaviEH runs the system's test suite against it, recording the passing and failing test cases. This task is also performed using the `maven-invoker` and `gradle-tooling-api`. Finally, at the end of Step (3), XaviEH provides a mutation analysis report of the system being analyzed.

In Step (4), XaviEH performs the test coverage analysis. To do this, XaviEH employs the JaCoCo[7] plug-in, a Java code coverage library for monitoring and tracking code coverage. For each system, XaviEH collects information to compute a suite of 23 test coverage metrics, as detailed in Chapter 4.4.

Finally, in Step (5), XaviEH summarizes the mutant and coverage analysis for the system under study. It generates a final report containing information on the number of generated and killed mutants, the mutation score, and the coverage metrics in CSV files.

4.3 Mutation Operators and Analysis

As discussed in Section 2.2, we used mutation testing to assess the effectiveness of the test suites under study on identifying defects related to EH code. Hence, we employed a set of EH-specific mutation operators proposed in previous studies (JI *et al.*, 2009; KUMAR *et al.*, 2011). Such mutation operators are based on real-world defects collected from empirical studies in open-source software. Thus, they mirror real defects introduced by developers. In total, we employed 7 mutation operators, as detailed in Table 2. The first 5 mutation operators (`CBR`, `CBI`, `CBD`, `PTL`, and `CRE`) were proposed by (JI *et al.*, 2009), and the final 2 operators (`FBD` and `TSD`) were proposed by (KUMAR *et al.*, 2011).

[6] https://spoon.gforge.inria.fr/
[7] https://www.eclemma.org/jacoco/

Table 2 – Mutation operators employed in this study. All operators are based on real-world defects from open-source systems.

Operator	Tranformation in the Code
CBR	Replaces the `catch` block with derived exception types according to the invoking exception hierarchy (JI *et al.*, 2009).
CBI	Creates complete `catch` modules to conceal all types of exceptions (JI *et al.*, 2009).
CBD	Deletes the whole `catch` block to propagate the thrown exceptions (JI *et al.*, 2009).
PTL	Brings into the `try` block, statements placed after the `try` block that reference variables inside the `try` block (JI *et al.*, 2009).
CRE	Re-throws the caught exceptions which are propagated to the upper modules (JI *et al.*, 2009).
FBD	Deletes the whole `finally` block to propagate the thrown exceptions (KUMAR *et al.*, 2011).
TSD	Deletes the `throw` statement that should raise an exception (KUMAR *et al.*, 2011).

4.4 Code Coverage Metrics

In this study, we adopted three different criteria to measure code coverage: instruction, branch, and method coverage. We have used the JaCoCo tool to compute the code coverage metrics. Instead of statements, JaCoCo computes the code coverage by analyzing bytecode instructions. Thus, we chose instruction coverage instead of statement coverage for compliance purposes.

We considered three sets of coverage metrics. In the first set, we computed the overall instruction, branch and method coverage, i.e. considering the system's entire code base. This is necessary for us to have a baseline of each system's general coverage, so that we can assess whether the EH code coverage presents any disparity when compared to the overall coverage. We detail the overall code coverage metrics in the first chapter of Table 3.

Next, the coverage metrics in the second set are tailored for EH code. It considers the instructions and branches inside `try`, `catch` and `finally` blocks, for instance. These are detailed in the second chapter of Table 3. Finally, the third set of coverage metrics is composed of complements for the EH-specific metrics.

Table 3 – Code coverage metrics computed by XaviEH. The first set of metrics are computed considering the system's entire code base. The second set of coverage metrics are specific for exception handling code. Finally, the third set is composed of complements to the exception handling coverage metrics.

Metric	Meaning
IC	The percentage of instructions exercised by the test suite.
BC	The percentage of branches exercised by the test suite.
MC	The percentage of methods exercised by the test suite.
Exception Handling Instruction Coverage (EH_IC)	The percentage of instruction in `try`, `catch`, and `finally` blocks plus all `throw` instructions exercised by the test suite.
Exception Handling Branch Coverage (EH_BC)	The percentage of branches in `try`, `catch`, and `finally` blocks exercised by the test suite.
Try Instruction Coverage (TRY_IC)	The percentage of instructions in `try` blocks exercised by the test suite.
Try Branch Coverage (TRY_BC)	The percentage of branches in `try` blocks exercised by the test suite.
Catch Instruction Coverage (CATCH_IC)	The percentage of instructions in `catch` blocks exercised by the test suite.
Catch Branch Coverage (CATCH_BC)	The percentage of branches in `catch` blocks exercised by the test suite.
Finally Instruction Coverage (FINALLY_IC)	The percentage of instructions in `finally` blocks exercised by the test suite.
Finally Branch Coverage (FINALLY_BC)	The percentage of branches in `finally` blocks exercised by the test suite.
Throw Instruction Coverage (THROW_IC)	The percentage of `throw` instructions exercised by the test suite.
Throws Method Coverage (THROWS_MC)	The percentage of methods with a `throws` clause in its signature exercised by the test suite.
Non-Exception Handling Instruction Coverage (NON_EH_IC)	The percentage of instructions exercised by the test suite that are not `throw` and not in `try`, `catch`, and `finally` blocks.
Non-Exception Handling Branch Coverage (NON_EH_BC)	The percentage of branches exercised by the test suite that are not in `try`, `catch`, and `finally` blocks.
Non-Try Instruction Coverage (NON_TRY_IC)	The percentage of instructions exercised by the test suite that are not in `try` blocks.
Non-Try Branch Coverage (NON_TRY_BC)	The percentage of branches exercised by the test suite that are not in `try` blocks.
Non-Catch Instruction Coverage (NON_CATCH_IC)	The percentage of instructions exercised by the test suite that are not in `catch` blocks.
Non-Catch Branch Coverage (NON_CATCH_BC)	The percentage of branches exercised by the test suite that are not in `catch` blocks.
Non-Finally Instruction Coverage (NON_FINALLY_IC)	The percentage of instructions exercised by the test suite that are not in `finally` blocks.
Non-Finally Branch Coverage (NON_FINALLY_BC)	The percentage of branches exercised by the test suite that are not in `finally` blocks.
Non-Throw Instruction Coverage (NON_THROW_IC)	The percentage of instructions exercised by the test suite that are not `throw`.
Non-Throws Method Coverage (NON_THROWS_MC)	The percentage of methods exercised by the test suite without a `throws` clause in its signature.

4.5 Preliminary Observation of the Libraries' Overall Coverage

To properly assess EH testing adequacy in terms of EH code coverage, we need to observe the systems' overall coverage to serve as a point of comparison. Otherwise, any high (or low) levels of EH code coverage that we observe in a software system may be due to the high (or low) levels of overall coverage in the system. Thus, this serves as baseline that we can take into account when drawing conclusions from our observations.

Fig. 2 presents boxplots depicting the distribution of overall instruction, branch, and method coverage, as detailed in Chapter 4.4 and Table 3. Note that the distributions were computed considering all the 27 studied libraries. The median values for IC, BC and MC are 82%, 78% and 83%, respectively. One must notice that apart from 2 outliers, all studied libraries tend to present coverage degrees in medium to high echelons, reaching more than 95% of coverage for some systems in all metrics. This indicates that the libraries under study present mature testing practices for the systems' overall source code.

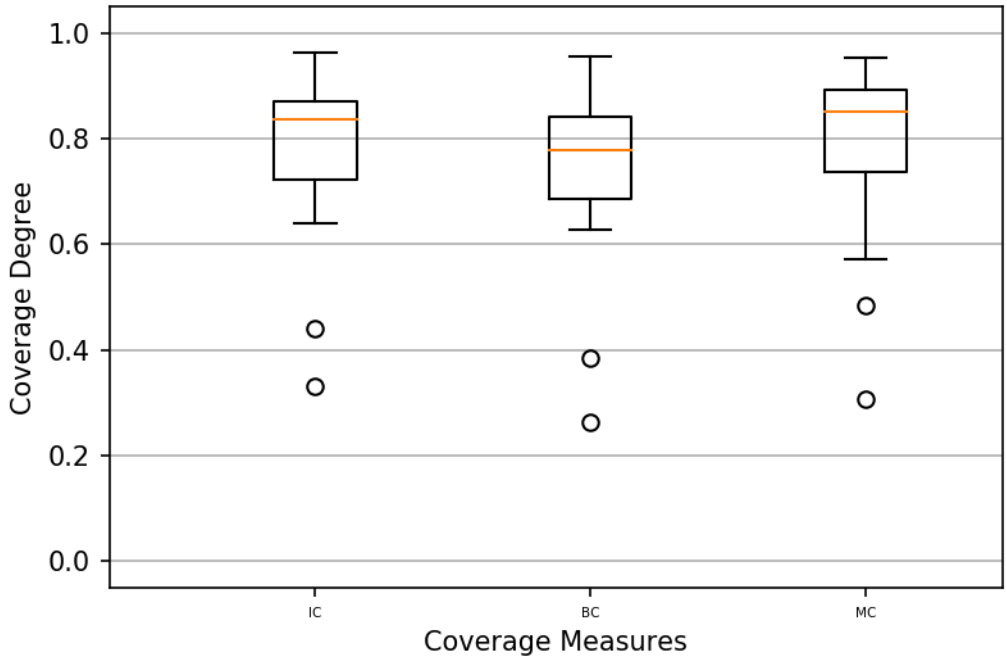

Figure 2 – Overall code coverage boxplots of the 27 studied libraries (IC, BC, and MC).

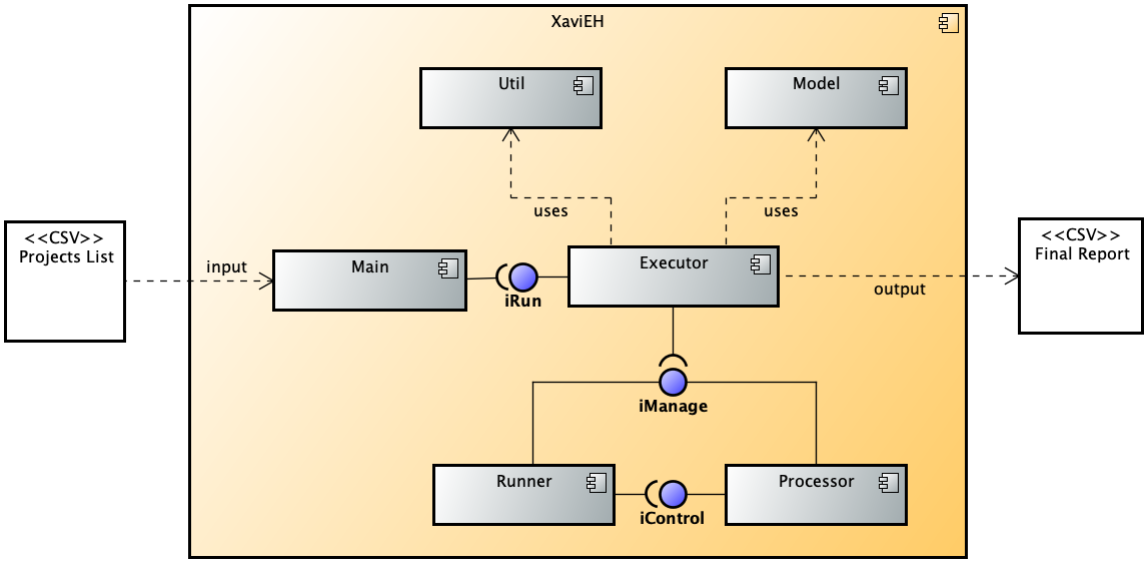

Figure 3 – XaviEH architecture, most important classes for operation.

4.6 XaviEH architecture

An important point in the development of systems is the architecture involved, it is possible to analyze the fundamental structure for the functioning of the developed tool. Figure 3 illustrates the architecture of XaviEH, its main points are explained in the points below:

– **Main Component**: is responsible for obtaining the user's input information, processing the input and making the correct execution of the classes **Executors** for the data processing

to take place correctly.

- **Classes Executors**: they are responsible for all the initialization of the customized classes of SPOON (**Runners** and **Processors**). It is also responsible for other uses of functions in the classes **Utilities** and classes **Models**. It also does all the analysis of the projects used for the study, among them, whether it is passing the tests or not and whether they have been used previously. Another great importance of this class is that it generates the reports of the jacoco, which are essential for the coverage analysis of the analyzed projects.
- **Runner Component**: is responsible for executing customized **Processors**, more specifically, initializes the SPOON API correctly, and executes the **Processors** passed by **Executors**. It also controls the extraction of the selected classes of projects that will be analyzed by **Processors**.
- **Processor Component**: is composed by responsible for the analyzes and modifications made in each class analyzed in the project, this is where the mutations and analyzes actually take place. Each **Processor** in the project is equivalent to a type of mutation (CBR, CBI, CBD, PTL, CRE, FBD and TSD).
- **Util Component**: is composed by classes used to support some task during the processes, among them, generation and writing of Extensible Markup Language (XML) or Comma Separated Values (CSV) and search for projects in GIT.
- **Model Component**: is composed by classes used as models for both input and output data in XaviEH processes.

5 STUDY RESULTS

In this chapter, we present the results obtained after applying our empirical strategy.

5.1 RQ1. What is the test coverage of EH code in long-lived Java libraries?

Summary of RQ1: `try` and `finally` blocks are largely more covered than `catch` blocks and `throw` statements, indicating that the test suites are struggling to raise and test exceptional behaviors in the programs.

Table 4 presents all computed code coverage metrics for all systems included in this study. Not all metrics could be computed for all systems. For instance, the BeanUtils library presents no `finally` block in its source code. As a result, all `finally`-related coverage metrics (FINALLY_IC and FINALLY_BC) could not be computed. We indicate with a '-' all cases in which a certain coverage metric could not be computed for a certain system.

The boxplots in Fig. 4 show the distribution of general coverage metrics for EH-related code. These are the EH coverage metrics that correspond to the overall coverage metrics displayed in Fig. 2. The coverage degree of `EH_IC` ranges from 55% to 74%, `EH_BC` ranges from 54% to 78%, and `THROWS_MC` ranges from 79% to 94%. Additionally, one should notice there exist libraries with 100% coverage for `EH_BC` and `THROWS_MC`.

We can draw interesting observations when comparing the EH-related coverage with the equivalent coverage metrics for the whole system displayed in Fig. 2. First, we observe a larger deviation in the adequacy of EH testing than in overall testing. This is depicted by how the boxplots for EH coverage tend to be less compact than the overall ones, which tend to indicate that the EH testing practices tend to be less mature than the overall testing ones. When considering instruction coverage for EH code, for example, we see systems with less than 40% of their EH instructions being covered, where the smallest overall instruction coverage is above 60%. Nevertheless, this is not always the case. We observed that a few systems reached 100% coverage EH methods, which did not occur for overall method coverage in any system.

We also plotted boxplots detailing the internal distribution of `EH_IC` (see Fig. 5) and `EH_BC` (see Fig. 6). Looking at Figure 5 and the data in Table 4, one can see that instructions in `try` and `finally` blocks have the best coverage degrees. In fact, they assume high levels of coverage if one consider the interquartile interval, ranging from 77% to 91% for `TRY_IC` and from 64% to 99% for `FINALLY_IC`. Differently, when considering the lowest quartile, the `throw`

Table 4 – Computed general code coverage measures per library.

Library	IC	BC	MC	EH_IC	EH_BC	TRY_IC	TRY_BC	CATCH_IC	CATCH_BC	FINALLY_IC	FINALLY_BC	THROW_IC	THROWS_MC
BCEL	0.44	0.38	0.48	0.31	0.33	0.38	0.35	0.13	0.25	0.15	0.11	0.01	0.70
BeanUtils	0.65	0.65	0.68	0.52	0.48	0.68	0.65	0.32	0.26	-	-	0.43	0.78
CLI	0.96	0.93	0.94	0.97	1.00	1.00	1.00	1.00	-	1.00	-	0.90	1.00
Codec	0.96	0.91	0.90	0.78	0.97	0.84	0.97	0.50	-	0.88	-	0.62	0.87
Collections	0.87	0.81	0.87	0.68	0.68	0.88	0.63	0.63	1.00	0.57	1.00	0.64	0.94
Compress	0.85	0.76	0.85	0.66	0.75	0.86	0.78	0.18	0.25	0.91	0.68	0.30	0.93
Configuration	0.88	0.84	0.91	0.76	0.74	0.81	0.77	0.45	0.42	1.00	0.70	0.72	0.93
DBCP	0.70	0.70	0.92	0.55	0.68	0.92	0.71	0.02	0.13	0.86	0.75	0.20	0.94
DbUtils	0.64	0.77	0.57	0.66	0.77	0.82	0.71	0.07	-	0.93	0.88	0.41	0.34
Digester	0.66	0.65	0.74	0.57	0.53	0.83	0.67	0.18	0.13	1.00	0.50	0.22	0.86
Email	0.72	0.67	0.81	0.65	0.57	0.77	0.67	0.59	0.75	0.29	0.15	0.24	0.87
Exec	0.72	0.63	0.74	0.42	0.54	0.43	0.52	0.27	0.50	0.75	0.67	0.31	0.65
FileUpload	0.80	0.76	0.67	0.69	0.69	0.79	0.79	0.24	-	0.81	0.50	0.44	0.68
Functor	0.82	0.66	0.90	0.23	-	-	-	-	-	-	-	0,23	-
IO	0.90	0.88	0.89	0.78	0.78	0.91	0.79	0.18	0.75	0.82	0.70	0.74	0.91
Lang	0.96	0.91	0.95	0.85	0.85	0.93	0.86	0.82	0.70	1.00	1.00	0.72	0.97
Math	0.92	0.85	0.87	0.65	0.72	0.92	0.91	0.53	0.53	0.73	-	0,59	0.89
Net	0.33	0.26	0.31	0.13	0.14	0.16	0.16	0.02	0.03	0.10	0.00	0,10	0,14
Pool	0.84	0.79	0.89	0.85	0.86	0.93	0.87	0.55	0.75	0.98	1.00	0.71	0.95
Proxy	0.82	0.80	0.85	0.55	0.43	0.52	0.43	1.00	-	-	-	0.62	1.00
Validator	0.86	0.76	0.81	0.52	0.54	0.75	0.60	0.17	0.30	0.00	-	0.43	0.86
Gson	0.84	0.79	0.85	0.67	0.66	0.83	0.63	0.47	1.00	1.00	1.00	0.34	0.97
Hamcrest	0.83	0.95	0.71	0.66	1.00	1.00	1.00	0.55	-	-	-	0.26	1.00
Jsoup	0.84	0.78	0.85	0.73	0.75	0.87	0.86	0.70	-	0.00	0.00	0.30	0.88
JUnit	0.86	0.83	0.88	0.66	0.64	0.83	0.68	0.45	0.45	0.93	0.88	0.54	0.94
Mockito	0.87	0.86	0.89	0.83	0.83	0.93	0.85	0.68	0.67	1.00	1.00	0.62	0.90
X-Stream	0.78	0.74	0.77	0.65	0.72	0.83	0.75	0.13	0.17	0.87	0.50	0.14	0.65

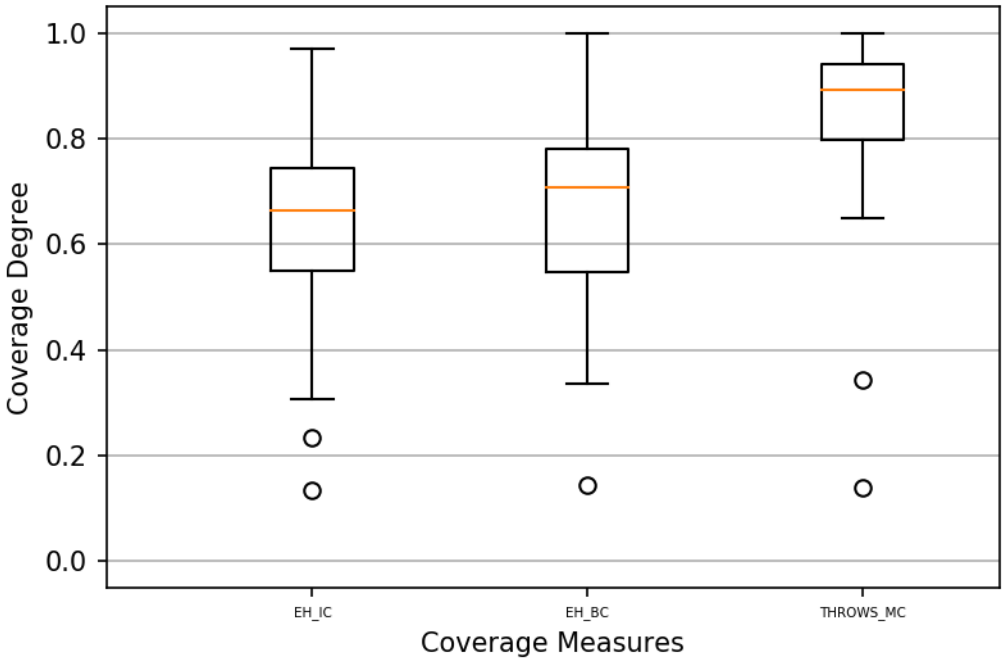

Figure 4 – Distribution of general EH-related code coverage metrics for the systems under study.

instructions and `catch` blocks have the worst coverage, with 25% and 17%, respectively. Hence, this suggests that `THROW_IC` and `CATCH_IC` are those the impact the general `EH_IC` the most.

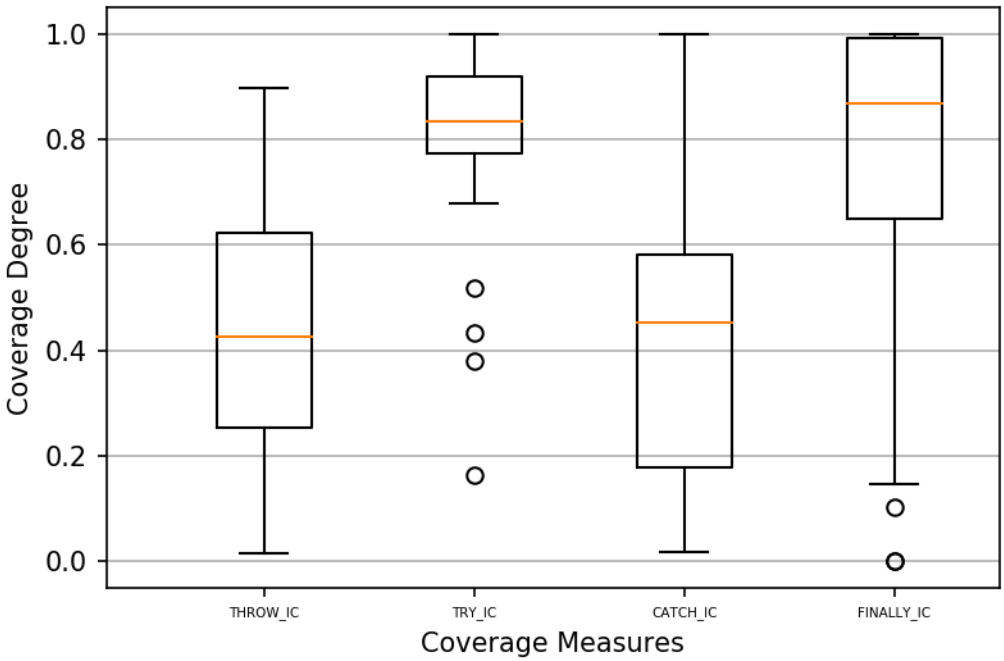

Figure 5 – EH instruction code coverage boxplots of studied libraries.

Looking at Fig. 6 and the data in Table 4, one can see that the branches in `try` and `finally` blocks have the better coverage when compared to the branches in `catch` blocks. In fact, if one consider the median of `TRY_BC` (73%) and `FINALLY_BC` (70%), one will see that about three-quarters of `CATCH_BC` is covered less than the median coverage of `TRY_BC` and `FINALLY_BC`. Thus, this suggests that `CATCH_BC` coverage is the one that impact most of the `EH_BC` coverage.

When analyzing the details of both instruction and branch coverage for EH code, we find a similar pattern, where `try` and `finally` blocks are largely more covered than `throw` instructions and `catch` blocks. This is an worrisome observation because `try` and `finally` blocks are always executed in non-exceptional behaviors of the system. Hence, we deduct that the test suites of the studied libraries are failing to raise exceptions. Thus, despite presenting high coverage for instruction and branches in the overall source code and EH code, the tests are still mostly exercising non-exceptional flows within the programs, where the exception behaviors are not being tested.

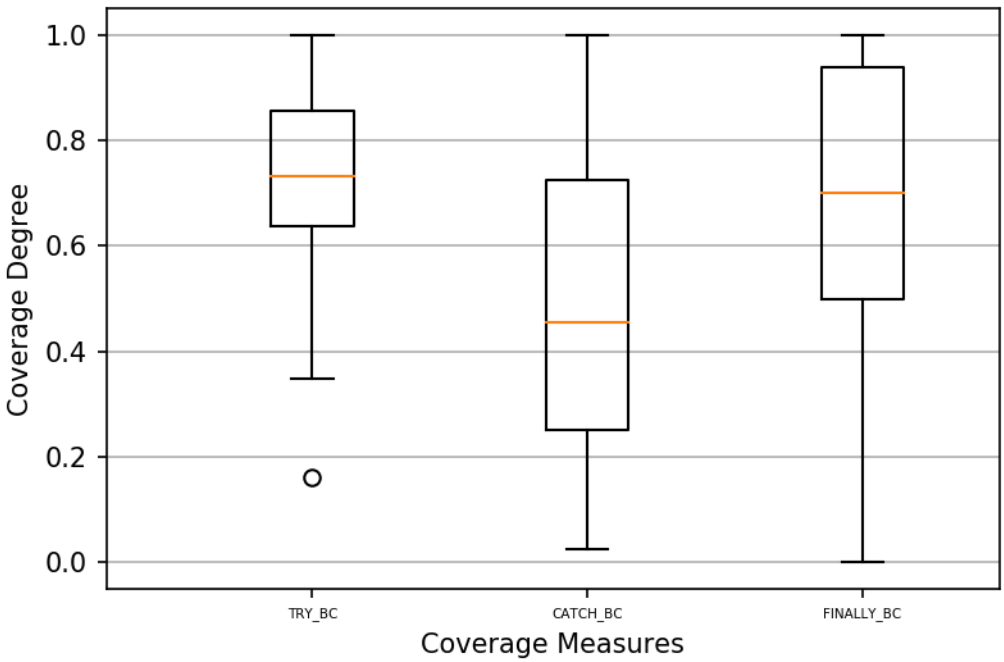

Figure 6 – EH branch code coverage boxplots of studied libraries.

5.2 RQ2. What is the difference between EH and non-EH code coverage in long-lived Java libraries?

Summary of RQ2: EH code is significantly less covered by test suites than non-EH code, especially regarding instructions and branches within `catch` blocks and `throw` instructions.

To answer this research question, we first computed the complementary code coverage metrics (see Table 3) for each studied library and summarize them in Table 5. Next, we employ two statistical tests to compare EH and non-EH code coverage values, in which we can verifying whether there are statistically significant differences between them. The first test is the Kolmogorov–Smirnov test (KS), and the second is the Mann-Whitney test (MW), as detailed next.

The KS is a two-sided test for the null hypothesis that two independent samples are drawn from the same continuous distribution. We test this null hypothesis by taking into account pairs of samples of non-EH and EH coverage metrics (see Tables 3, 4 and 5). Consider instruction coverage, for example, where we abbreviate it to simply *A* for brevity. We measured both `EH_IC` and `NON_EH_IC` for all 27 libraries under study. We formulate our null hypothesis as $\mathscr{H}_0^A$: `NON_EH_IC` = `EH_IC`. In case this KS null hypothesis cannot be rejected, we assume that non-EH and EH code coverage measures have the same distribution, i.e., there is no statistical difference in instruction coverage for EH and non-EH code when considering all libraries. However, in case the KS null hypothesis is rejected, we assume the alternative hypothesis $\mathscr{H}_1^A$: `NON_EH_IC` $\neq$ `EH_IC`, which indicates statistical difference in instruction coverage between EH and non-EH code.

In case statistical difference is indicated by the KS test, we can employ the MW test to assert whether the coverage values in non-EH code are higher than the coverage values in EH code, or vice-versa. Consider the instruction coverage metric, for example. The MW test considers the null hypothesis $\mathscr{H}_0^A$: `NON_EH_IC` $>$ `EH_IC`. If the MW null hypothesis cannot be rejected, we assume that the instruction coverage of non-EH code is significantly greater than the instruction coverage in EH code. Otherwise, we assume MW's alternative hypothesis ($\mathscr{H}_1^A$: `NON_EH_IC` $<$ `EH_IC`), which indicates that instruction coverage in EH code is significantly greater than in non-EH code. Table 6 presents the statistical tests results for all coverage metrics with the significance level of $\alpha < 0.05$.

To enhance this analysis, in Fig. 7, we depict boxplots for the instruction, branch and method coverage for both non-EH and EH code. When comparing the boxplots, one can see that

Table 5 – Computed complementary code coverage measures per library.

Library	NON_EH_IC	NON_EH_BC	NON_TRY_IC	NON_TRY_BC	NON_CATCH_IC	NON_CATCH_BC	NON_FINALLY_IC	NON_FINALLY_BC	NON_THROW_IC	NON_THROWS_MC
BCEL	0.45	0.04	0.44	0.39	0.44	0.38	0.44	0.39	0.45	0.47
BeanUtils	0.66	0.16	0.64	0.65	0.66	0.67	0.65	0.65	0.65	0.66
CLI	0.96	0.76	0.96	0.93	0.96	0.93	0.96	0.93	0.96	0.93
Codec	0.97	0.54	0.96	0.91	0.96	0.91	0.96	0.91	0.97	0.91
Collections	0.87	0.41	0.87	0.81	0.87	0.81	0.87	0.81	0.87	0.87
Compress	0.85	0.28	0.85	0.76	0.85	0.76	0.85	0.76	0.85	0.82
Configuration	0.89	0.38	0.88	0.84	0.88	0.84	0.88	0.84	0.88	0.91
DBCP	0.78	0.13	0.64	0.70	0.80	0.71	0.69	0.70	0.71	0.89
DbUtils	0.63.	0.10	0.62	0.78	0.65	0.77	0.63	0.77	0.64	0.84
Digester	0.67	0.14	0.65	0.65	0.67	0.65	0.66	0.65	0.67	0.72
Email	0.74	0.19	0.72	0.67	0.72	0.67	0.73	0.69	0.74	0.77
Exec	0.78	0.21	0.75	0.64	0.73	0.63	0.72	0.63	0.73	0.75
FileUpload	0.81	0.25	0.80	0.76	0.80	0.76	0.80	0.77	0.80	0.67
Functor	0.83	0.38	0.82	0.66	0.82	0.66	0.82	0.66	0.83	0.90
IO	0.92	0.50	0.90	0.89	0.91	0.88	0.90	0.88	0.91	0.88
Lang	0.96	0.75	0.96	0.91	0.96	0.91	0.96	0.91	0.96	0.95
Math	0.93	0.41	0.92	0.85	0.93	0.85	0.92	0.85	0.93	0.87
Net	0.36	0.03	0.35	0.28	0.34	0.27	0.33	0.26	0.33	0.40
Pool	0.84	0.26	0.83	0.78	0.85	0.79	0.84	0.79	0.85	0.88
Proxy	0.84	0.22	0.84	0.82	0.82	0.80	0.82	0.80	0.83	0.84
Validator	0.87	0.31	0.86	0.76	0.87	0.76	0.86	0.76	0.87	0.81
Gson	0.85	0.36	0.84	0.80	0.84	0.79	0.83	0.79	0.85	0.83
Hamcrest	0.84	0.24	0.83	0.95	0.84	0.95	0.83	0.95	0.84	0.71
Jsoup	0.84	0.35	0.84	0.78	0.84	0.78	0.84	0.78	0.84	0.85
JUnit	0.87	0.30	0.86	0.83	0.87	0.83	0.86	0.83	0.86	0.88
Mockito	0.88	0.34	0.87	0.86	0.87	0.86	0.87	0.86	0.88	0.89
X-Stream	0.79	0.25	0.77	0.74	0.78	0.75	0.78	0.74	0.79	0.78

Table 6 – Summary of hypothesis statement and the statistics test results.

KS Hypothesis	p-value	MW Hypothesis	p-value
$\mathscr{H}_0^A$: NON_EH_IC = EH_IC () $\mathscr{H}_1^A$: NON_EH_IC ≠ EH_IC ()	9.8×10^{-5}	$\mathscr{H}_0^A$: NON_EH_IC > EH_IC () $\mathscr{H}_1^A$: NON_EH_IC < EH_IC ()	6.2×10^{-5}
$\mathscr{H}_0^B$: NON_EH_BC = EH_BC () $\mathscr{H}_1^B$: NON_EH_BC ≠ EH_BC ()	2.1×10^{-8}	$\mathscr{H}_0^B$: NON_EH_BC > EH_BC () $\mathscr{H}_1^B$: NON_EH_BC < EH_BC ()	3.0×10^{-7}
$\mathscr{H}_0^C$: NON_THROWS_MC = THROWS_MC () $\mathscr{H}_1^C$: NON_THROWS_MC ≠ THROWS_MC ()	3.0×10^{-2}	$\mathscr{H}_0^C$: NON_THROWS_MC > THROWS_MC () $\mathscr{H}_1^C$: NON_THROWS_MC < THROWS_MC ()	2.0×10^{-2}
$\mathscr{H}_0^D$: NON_THROW_IC = THROW_IC () $\mathscr{H}_1^D$: NON_THROW_IC ≠ THROW_IC ()	1.8×10^{-7}	$\mathscr{H}_0^D$: NON_THROW_IC > THROW_IC () $\mathscr{H}_1^D$: NON_THROW_IC < THROW_IC ()	1.5×10^{-7}
$\mathscr{H}_0^E$: NON_TRY_IC = TRY_IC () $\mathscr{H}_1^E$: NON_TRY_IC ≠ TRY_IC ()	8.0×10^{-1}	$\mathscr{H}_0^E$: NON_TRY_IC > TRY_IC (?) $\mathscr{H}_1^E$: NON_TRY_IC < TRY_IC (?)	N/A
$\mathscr{H}_0^F$: NON_CATCH_IC = CATCH_IC () $\mathscr{H}_1^F$: NON_CATCH_IC ≠ CATCH_IC ()	2.9×10^{-7}	$\mathscr{H}_0^F$: NON_CATCH_IC > CATCH_IC () $\mathscr{H}_1^F$: NON_CATCH_IC < CATCH_IC ()	2.1×10^{-6}
$\mathscr{H}_0^G$: NON_FINALLY_IC = FINALLY_IC () $\mathscr{H}_1^G$: NON_FINALLY_IC ≠ FINALLY_IC ()	1.6×10^{-1}	$\mathscr{H}_0^G$: NON_FINALLY_IC > FINALLY_IC (?) $\mathscr{H}_1^G$: NON_FINALLY_IC < FINALLY_IC (?)	N/A
$\mathscr{H}_0^H$: NON_TRY_BC = TRY_BC () $\mathscr{H}_1^H$: NON_TRY_BC ≠ TRY_BC ()	5.4×10^{-1}	$\mathscr{H}_0^H$: NON_TRY_BC > TRY_BC (?) $\mathscr{H}_1^H$: NON_TRY_BC < TRY_BC (?)	N/A
$\mathscr{H}_0^I$: NON_CATCH_BC = CATCH_BC () $\mathscr{H}_1^I$: NON_CATCH_BC ≠ CATCH_BC ()	8.8×10^{-4}	$\mathscr{H}_0^I$: NON_CATCH_BC > CATCH_BC () $\mathscr{H}_1^I$: NON_CATCH_BC < CATCH_BC ()	3.2×10^{-4}
$\mathscr{H}_0^J$: NON_FINALLY_BC = FINALLY_BC () $\mathscr{H}_1^J$: NON_FINALLY_BC ≠ FINALLY_BC ()	1.6×10^{-1}	$\mathscr{H}_0^J$: NON_FINALLY_BC > FINALLY_BC (?) $\mathscr{H}_1^J$: NON_FINALLY_BC < FINALLY_BC (?)	N/A

the EH instruction coverage (EH_IC) is lower than non-EH instruction coverage (NON_EH_IC). This perception is confirmed by the statistical tests results that reject the KS null hypothesis $\mathscr{H}_0^A$: NON_EH_IC = EH_IC and did not reject the MW null hypothesis $\mathscr{H}_0^A$: NON_EH_IC > EH_IC. This indicates that not only the instruction coverage of EH and non-EH code are statistically different but also that non-EH code is statistically more covered than EH code.

On the other hand, when considering branch coverage, EH code (EH_BC) seems to be more covered than non-EH code (NON_EH_BC). Indeed, this perception is confirmed by the statistical tests results that reject the KS null hypothesis $\mathscr{H}_0^B$: NON_EH_BC = EH_BC and also reject the MW null hypothesis $\mathscr{H}_0^B$: NON_EH_BC > EH_BC. This is a counter intuitive observation given the results previously observed in our study. We address this during our study' discussion (see Chapter 6).

Finally, different from instruction and branch coverage, the values of method coverage for EH code (THROWS_MC) and non-EH code (NON_THROWS_MC) are visually similar. However, this perception is not confirmed by the statistical tests that reject both the KS null hypothesis

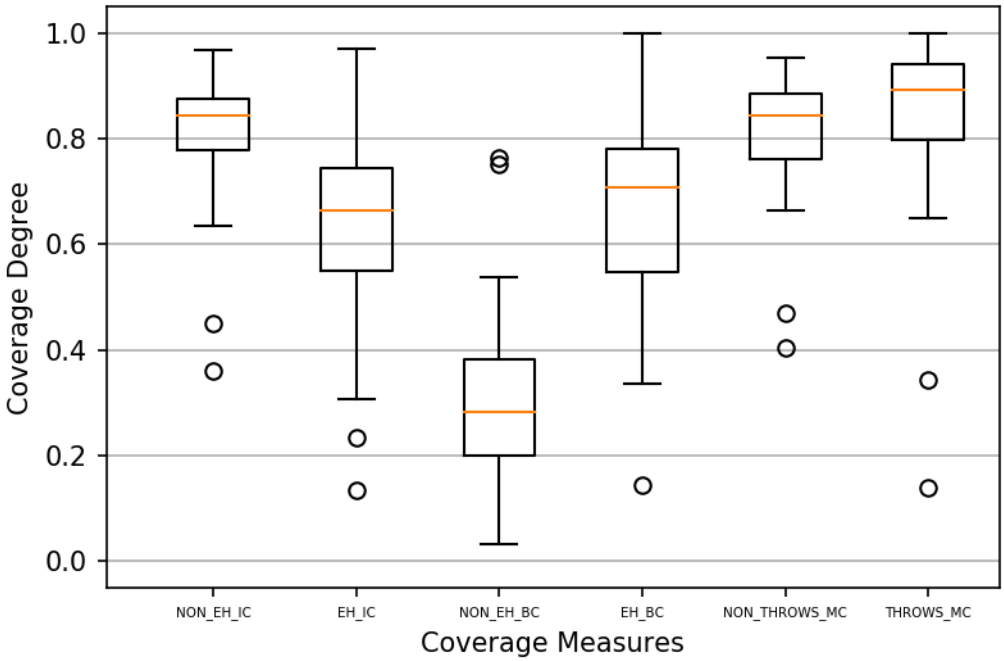

Figure 7 – Overall EH and non-EH code coverage boxplots of studied libraries.

$\mathscr{H}_0^C$: NON_THROWS_MC = THROWS_MC and the MW null hypothesis $\mathscr{H}_0^C$: NON_THROWS_MC > THROWS_MC. This indicates that methods without a `throws` clause are significantly less covered than methods with a `throws` clause. Implications for this finding are also addressed in our discussion chapter.

Fig. 8 presents boxplots detailing instruction coverage metrics for non-EH and EH code. It depicts coverage values for `throw` instructions and instructions in `try`, `catch` and `finally` blocks, respectively. When comparing the boxplots of THROW_IC and NON_THROW_IC metrics, one may notice that the `throw` instructions are less covered than the non-`throw` instructions. This perception is confirmed by the statistical tests results that reject the KS null hypothesis $\mathscr{H}_0^D$: NON_THROW_IC = THROW_IC and accept the MW null hypothesis $\mathscr{H}_0^D$: NON_THROW_IC > THROW_IC. This indicates that even though the systems under study present a fairly high level of instruction coverage (see Fig. 2), the instructions that actually raise exceptions are not well covered.

When comparing the boxplots of CATCH_IC and NON_CATCH_IC coverage, one can see that the instructions inside `catch` blocks are covered less than the instructions outside `catch` blocks. Once again, the statistical tests results confirm this perception by rejecting the KS null hypothesis $\mathscr{H}_0^F$: NON_CATCH_IC = CATCH_IC and accepting the MW null hypothe-

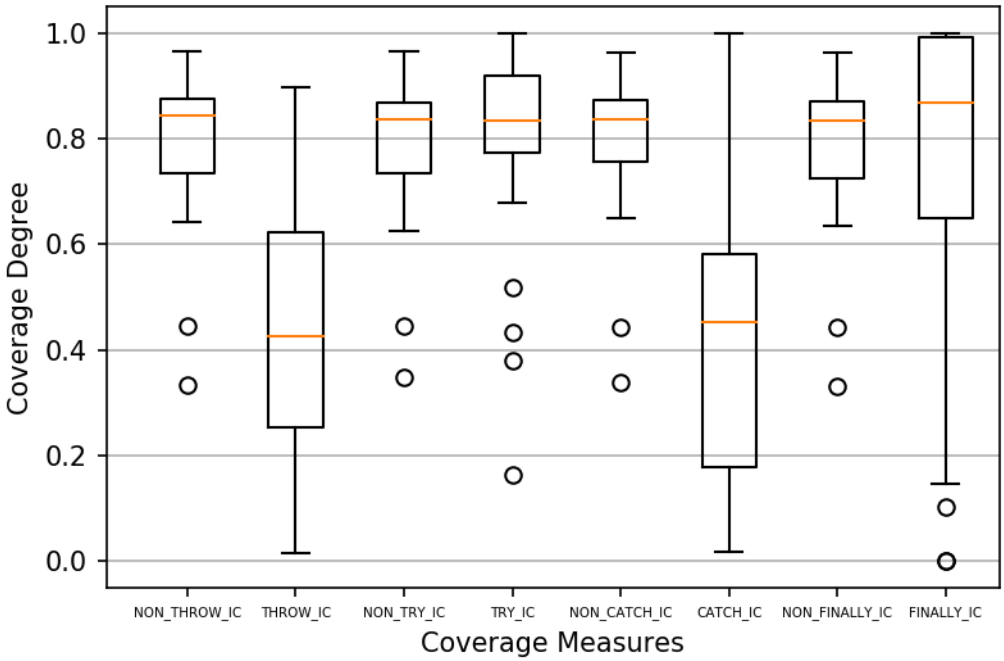

Figure 8 – The EH and non-EH instruction coverage boxplots of studied libraries.

sis $\mathscr{H}_0^F$: NON_CATCH_IC $>$ CATCH_IC. This represents additional evidence that EH code is considerably less covered than non-EH code.

However, when we look at the coverage inside `try` and `finally` blocks and their counterparts (i.e., the coverage of instructions outside `try` and `finally` blocks) we realize they are similar. This perception is confirmed by the statistical tests when both the KS and MW null hypotheses ($\mathscr{H}_0^E$: NON_TRY_IC $=$ TRY_IC and $\mathscr{H}_0^G$: NON_FINALLY_IC $=$ FINALLY_IC) are accepted. Since `try` and `finally` blocks are commonly executed when no exceptional behavior is exercised, this observation corroborates with previous findings that code that handle exceptions are not properly tested.

The boxplots of Fig. 9 shows the branch coverage distribution of EH and non-EH code. When comparing the boxplots of CATCH_BC and NON_CATCH_BC, one must notice that branches inside `catch` blocks are less covered than branches outside `catch` blocks. Once more, the statistical test results confirm this perception by rejecting the KS null hypothesis $\mathscr{H}_0^I$: NON_CATCH_BC $=$ CATCH_BC and accepting the MW null hypothesis $\mathscr{H}_0^I$: NON_CATCH_BC $>$ CATCH_BC.

However, when we compare the coverage of branches inside `try` and `finally` blocks with their counterparts (the coverage of branches outside `try` and `finally` blocks) we

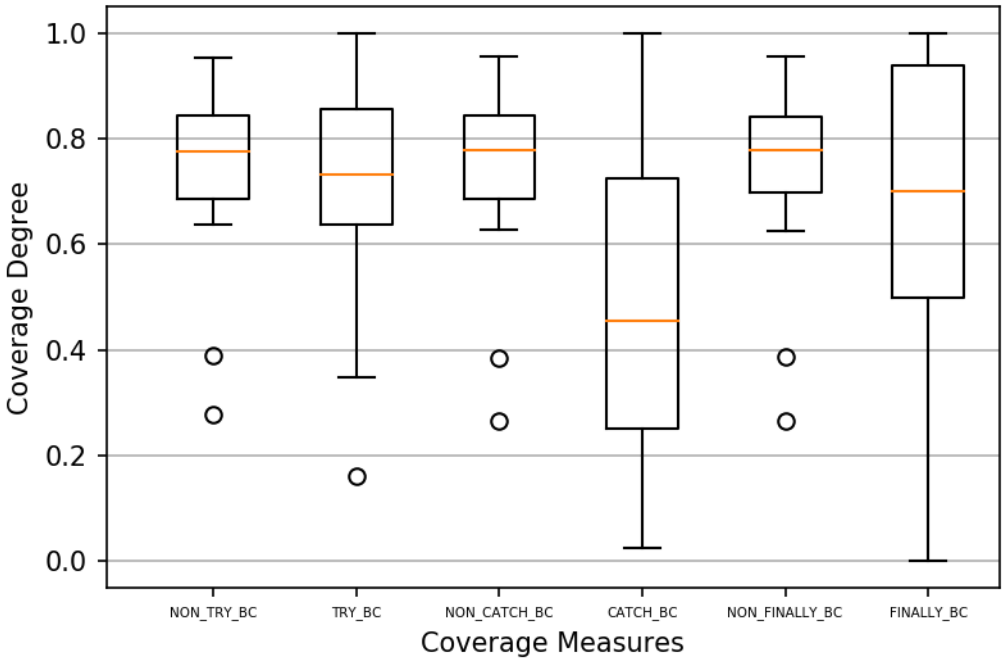

Figure 9 – The EH and non-EH branch coverage boxplots of studied libraries.

realize they are similar. This perception is also confirmed by the statistical tests when both the KS and MW null hypotheses ($\mathscr{H}_0^H$: `NON_TRY_BC` = `TRY_BC` and $\mathscr{H}_0^J$: `NON_FINALLY_BC` = `FINALLY_BC`) are accepted. Once again, all findings regarding branch coverage add to the observation that code which raises and handle exceptions are statistically less covered than regular code.

5.3 RQ3. What is the effectiveness of EH testing in long-lived Java libraries?

Summary of RQ3: The systems under study present effective test suites for EH code, where more than 78% of the defects being identified. However, the systems present difficulties in identifying defects in `finally` blocks.

In this study, we employ mutation testing to assess the effectiveness of EH testing in the libraries under study, as detailed in Section 4.2. In Table 7, we present results of the mutation testing analysis we performed. For each mutation operator (see Chapter 4.3), we show the number of mutants killed by the test suite, the number of mutants left alive, and the mutation score.

Furthermore, in Fig. 10, we present boxplots showing the mutation score distribution for all libraries and each mutation operator. Note that we computed the distributions considering

Table 7 – Details of the mutation testing analysis for each library. Column L indicates the number of live mutants, column D indicates the number of killed mutants, and S indicates the mutation score.

Library	CBI			CBD			CRE			FBD			PTL			CBR			TSD			OVERALL		
	L	D	S	L	D	S	L	D	S	L	D	S	L	D	S	L	D	S	L	D	S	L	D	S
BCEL	1	6	0.86	23	113	0.83	23	113	0.83	4	1	0.20	0	3	1.00	1	11	0.92	110	296	0.73	162	543	0.77
BeanUtils	1	9	0.90	42	84	0.67	24	102	0.81	0	0	0.00	0	8	1.00	0	13	1.00	185	179	0.49	252	395	0.61
CLI	0	0	0.00	1	10	0.91	1	10	0.91	0	1	1.00	0	0	0.00	0	0	0.00	1	28	0.97	3	49	0.94
Codec	0	0	0.00	3	18	0.86	3	18	0.86	8	0	0.00	0	0	0.00	0	0	0.00	42	55	0.57	56	91	0.62
Collections	0	0	0.00	4	24	0.86	4	24	0.86	1	0	0.00	0	1	1.00	0	0	0.00	159	566	0.78	168	615	0.79
Compress	0	5	1.00	13	057	0.81	14	56	0.80	36	3	0.08	0	3	1.00	9	0	0.00	216	209	0.49	288	333	0.54
Configuration	1	6	0.86	3	125	0.98	3	125	0.98	2	93	0.98	0	4	1.00	0	4	1.00	21	264	0.93	30	621	0.95
DBCP	3	7	0.70	617	163	0.21	611	169	0.22	6	17	0.74	0	8	1.00	0	7	1.00	79	200	0.72	1316	571	0.30
DbUtils	0	0	0.00	11	20	0.65	11	20	0.65	3	17	0.85	0	1	1.00	0	0	0.00	30	16	0.35	55	74	0.57
Digester	4	1	0.20	22	41	0.65	22	41	0.65	5	2	0.29	0	3	1.00	1	3	0.75	79	31	0.28	133	122	0.48
Email	0	3	1.00	2	027	0.93	3	26	0.90	8	1	0.11	0	1	1.00	0	3	1.00	2	72	0.97	15	133	0.90
Exec	0	2	1.00	6	15	0.71	4	17	0.81	3	2	0.40	0	0	0.00	0	1	1.00	0	29	1.00	13	66	0.84
FileUpload	0	0	0.00	7	15	0.68	6	16	0.73	5	1	0.17	0	0	0.00	0	0	0.00	26	24	0.48	44	56	0.56
Functor	0	0	0.00	0	0	0.00	0	0	0.00	0	0	0.00	0	0	0.00	0	0	0.00	90	25	0.22	90	25	0.22
IO	0	2	1.00	40	38	0.49	46	32	0.41	2	6	0.75	0	1	1.00	0	2	1.00	37	255	0.87	125	336	0.73
Lang	0	2	1.00	13	058	0.82	15	56	0.79	1	4	0.80	0	3	1.00	0	1	1.00	88	292	0.77	117	416	0.78
Math	6	6	0.50	21	094	0.82	20	095	0.83	4	0	0.00	0	5	1.00	4	5	0.56	414	1080	0.72	469	1285	0.73
Net	1	5	0.83	31	129	0.81	26	134	0.84	9	15	0.63	1	5	0.83	0	3	1.00	47	112	0.70	115	403	0.78
Pool	1	3	0.75	8	057	0.88	8	057	0.88	46	33	0.42	0	7	1.00	0	1	1.00	15	64	0.81	78	222	0.74
Proxy	0	0	0.00	0	23	1.00	1	22	0.96	0	0	0.00	0	0	0.00	0	0	0.00	0	36	1.00	1	81	0.99
Validator	0	1	1.00	10	30	0.75	11	29	0.73	1	0	0.00	0	3	1.00	0	2	1.00	18	50	0.74	40	115	0.74
Gson	0	3	1.00	6	49	0.89	5	50	0.91	1	4	0.80	0	1	1.00	0	4	1.00	26	196	0.88	38	307	0.89
Hamcrest	0	0	0.00	1	11	0.92	1	11	0.92	0	0	0.00	0	0	0.00	0	0	0.00	7	12	0.63	9	34	0.79
Jsoup	0	0	0.00	2	30	0.94	2	30	0.94	1	2	0.67	0	3	1.00	0	0	0.00	11	35	0.76	16	100	0.86
JUnit	0	5	1.00	7	083	0.92	9	81	0.90	6	11	0.65	0	2	1.00	0	3	1.00	20	81	0.80	42	266	0.86
Mockito	1	10	0.91	4	076	0.95	7	73	0.91	7	13	0.65	0	0	0.00	0	0	0.00	25	211	0.89	44	383	0.90
X-Stream	1	2	0.67	36	196	0.84	32	200	0.86	8	11	0.58	2	19	0.90	1	1	0.50	160	301	0.65	240	730	0.75

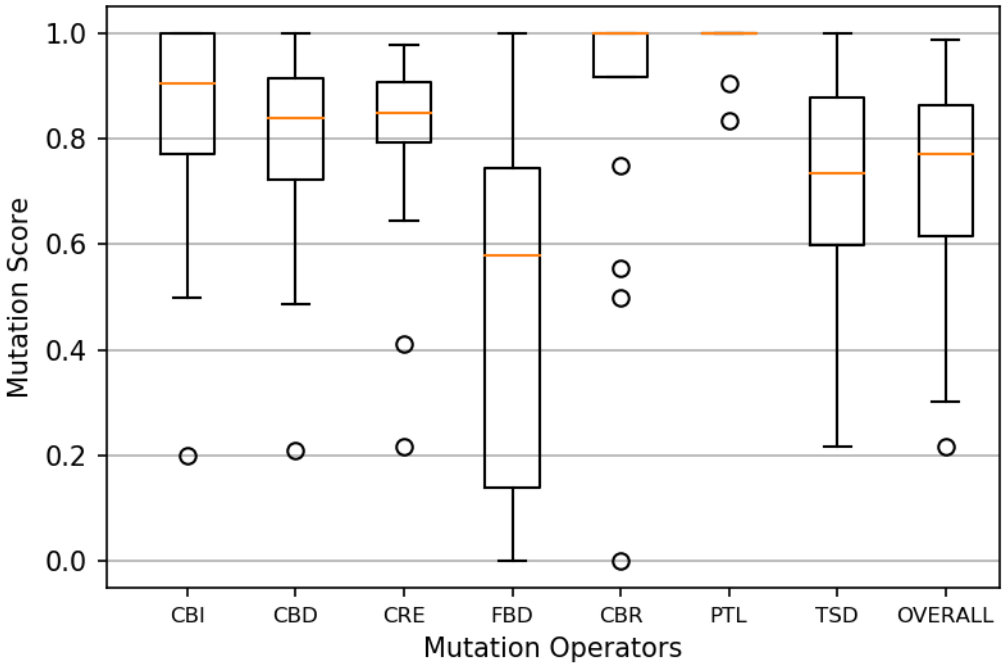

Figure 10 – Mutation scores distribution bloxpots.

only the mutation scores of libraries in which we were able to generate at le ast one mutant using the mutation operator associated with the boxplot. In this study, we generated a total of 12,331 software mutants as follows: 98 (CBI), 2,519 (CBD), 2,519 (CRE), 404 (FBD), 84 (PTL), 80 (CBR), and 6,627 (TSD).

When looking at both the table and figure, one must notice that the systems under study achieved mean and median values mutation score above 70% for all but one mutation operator (FBD), which is considerably high when compared with other studies in the literature (Reales *et al.*, 2014; GOPINATH *et al.*, 2014; INOZEMTSEVA; HOLMES, 2014b). When taking into account all mutation operators, the median mutation score achieved is 78%. This indicates that the test suites in the studied libraries managed to detect a median of 78% of artificially injected defects. Operators such as CBR and PTL, for example, present median mutation scores of 100%, indicating that the test suites of most of the systems under study identified all bugs related to wrongly declared exceptions in `catch` blocks and wrongly placed instructions in `try` blocks.

Nevertheless, this is not the case for all mutation operators. We observe a median mutation score of 59% for the FBD operator, reaching even 0% for some systems. This indicates that the libraries under study struggle in identifying defects in `finally` blocks. This is an interesting observation because we showed in the previous research question that `finally`

blocks are highly covered. Hence, although being able to exercise EH code in `finally` blocks, the test suites have difficulties in actually identifying defects in them.

It is important to notice that not all mutation operators generated a similar number of mutants. On the contrary, there are large differences between operators, such as `TSD` generating a total of 6,627 mutants for all systems and `CBR` generating only 80 mutants overall. However, there seems to be no relationship between the number of mutants generated as the mutation score achieved. For example, two operators with a small number of generated mutants, such as `FBD` and `PTL`, represent the operators with smallest and highest median of mutation score, respectively. The relationship between number of mutants and the respective effectiveness of the test suite for this type of defects is still open for investigation.

5.4 RQ4. To what extent are there EH bugs that are statistically harder to detect by test suites of long-lived Java libraries?

Summary of RQ4: There are EH bugs that are statistically harder to detect than others. In specfic, EH bugs of types `FBD`, `TSD`, `CRE`, and `CBD` are more difficult to detect than EH bugs of type `PTL`.

To properly answer this research question, we employed a statistical test to verify whether there is any significant difference between the effectiveness of the studied libraries' test suites in detecting different types of artificially injected EH bugs (i.e., EH mutants). We used the Friedman test (FRIEDMAN, 1940), which aims at quantifying the consistency of the results obtained by a test suite when applied over several types of EH bugs, generated using different type of mutation operators, according to their average performance rankings (for each test suite, the highest mutation score getting rank 1, the second-highest rank 2 and so on). In the current setting, the null hypothesis states that there is no statistical difference in detecting different types of EH bugs. If the Friedman null hypothesis is rejected, a post-hoc test must be applied to identify what type of EH bug is significantly easier/harder to detect than others. For this purpose, we adopt the post-hoc Nemenyi test (DEMsAR, 2006).

To ensure that the Friedman's test will yield significant results, the data points cannot present missing values. Since XaviEH could not generate mutants for a few operators in some libraries (see Table 7), we selected for this analysis only the test suites of libraries that XaviEH could generate mutants for all mutation operators, which represents a total of 15 studied libraries. Despite losing data points for this analysis, we can still observe statistically significant results

Table 8 – The ranks and average rank of mutation scores.

Library	CBI	CBD	CRE	FBD	PTL	CBR	TSD
BCEL	3.0	4.5	4.5	7.0	1.0	2.0	6.0
Compress	1.5	3.0	4.0	6.0	1.5	7.0	5.0
Configuration	7.0	4.5	4.5	3.0	1.5	1.5	6.0
DBCP	5.0	7.0	6.0	3.0	1.5	1.5	4.0
Digester	7.0	3.5	3.5	5.0	1.0	2.0	6.0
Email	2.0	5.0	6.0	7.0	2.0	2.0	4.0
IO	2.0	6.0	7.0	5.0	2.0	2.0	4.0
Lang	2.0	4.0	6.0	5.0	2.0	2.0	7.0
Math	6.0	3.0	2.0	7.0	1.0	5.0	4.0
Net	3.5	5.0	2.0	7.0	3.5	1.0	6.0
Pool	6.0	3.5	3.5	7.0	1.5	1.5	5.0
Validator	2.0	4.0	6.0	7.0	2.0	2.0	5.0
Gson	2.0	5.0	4.0	7.0	2.0	2.0	6.0
JUnit	2.5	4.0	5.0	7.0	2.0	2.0	6.0
X-Stream	4.0	3.0	2.0	6.0	1.0	7.0	5.0
Average Rank	**3.7**	**4.3**	**4.4**	**5.9**	**1.7**	**2.7**	**5.3**

because Friedman's test guidelines state that p-values are reliable for more than 6 measurements (libraries test suites in our study).

For each libraries' test suite, we used the set of all 7 EH mutation operators and ranked them according to their mutation scores. Consider the `BCEL` library, for example. The `PTL` operator presented the highest mutation score, which indicates that this was the easiest type of EH bug to detect in this library, yielding a rank 1. Similarly, the `FBD` operator received the rank 7 because it presented the lowest mutation score for all operators in the `BCEL` library. Next, we averaged the rankings for all mutation operators and produced the final average ranking. We present all computed rankings in Table 8.

According to the Friedman test, the average ranking difference is significant with p-value $= 1.2 \times 10^{-4}$. Hence, we attest that there exists types of EH bugs that are statistically harder to identify by the test-suites under study. Next, we employed Nemenyi's post-hoc test, which showed a Critical Difference of `Critical Difference (CD)` = 2.32. In this context, the performance of two mutation operators is said to be significantly different if their average ranking differ by at least the `CD` level. The `CD` metric is computed using Equation 5.1, where k is the number of mutation operators, N is the number of test suites (libraries), and q_α is a pre-calculated critical value that one must pick up from a reference table by observing the value of k and the

confidence interval (α). Thus, for our study $k = 7$, $N = 15$, $\alpha = 0.05$, and $q_{0.05} = 2.948$.

$$\texttt{CD} = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \tag{5.1}$$

Based on these results, we can conclude that the FBD and TSD mutation operators generate EH bugs that are statistically more difficult to detect than EH bugs generated by the PTL and CBR mutation operators. Additionally, the CRE and CBD operators generate EH bugs that are significantly harder to detect than EH bugs generated by the PTL operator. Even though we observe differences in the ranking between FBD, TSD, CRE, and CBD, we cannot ascertain significant statistical difference between them. This indicates that, according to our empirical study, these are equally the most difficult types of EH bugs to detect.

6 DISCUSSION

In this chapter, we sum up the most important findings of our empirical study and discuss their implications. Finally, we briefly discuss on how XaviEH could be used in practice.

6.1 On the Adequacy of EH Testing

In RQ1-2, we present empirical and statistical evidence that EH code is less covered than regular code in the systems under study. Moreover, we show that within coverage of EH code, instructions and branches inside `catch` blocks and `throw` instructions are statistically less covered than instructions and branches in `try` and `finally` blocks. This indicates that not only EH code (statements in `catch` blocks) is not properly covered by test suites but also that these suites are not able to reach the code parts responsible to raise exceptions (`throw` statements). In fact, we have computed the Spearman's correlation between `throw` instruction coverage (`THROW_IC`) and `catch` blocks' instruction (`CATCH_IC`) and branch (`CATCH_BC`) coverage. The result shows a strong correlation in both cases with $\rho = 0.582664$ (`THROW_IC` and `CATCH_IC`) and $\rho = 0.674882$ (`THROW_IC` and `CATCH_BC`). This is an worrisome finding since the main goal of EH testing should be raising exceptions in order to test exceptional behaviors in the programs. For without even raising exceptions, developers cannot test whether the code to handle the exception is correct.

Hence, our study shows that developers need better support in designing test cases that exercise exceptional behaviors. In addition to creating guidelines, this may be accomplished through search-based testing (MCMINN, 2004), where optimization algorithms and metaheuristics are used to automatically generate test cases according to a certain objective function. In this case, one could set the coverage of `throw` instructions and branches and instructions inside `catch` blocks as a goal. To the best of our knowledge, there is few work in this direction (ROMANO *et al.*, 2011).

6.2 On the Effectiveness of EH Testing

RQ3-4 show that despite not properly covering EH code, the test suites of the libraries under study are surprisingly effective in identifying artificially injected faults (EH mutants). Most of the systems presented mutation scores of more than 70% for most mutation operators. However, this was not the case for all operators. In fact, we showed that there do exist

statistically harder types of EH bugs to identify. These are commonly related to mutations in `throw` statements and `catch` and `finally` blocks. This is an interesting finding that corroborate with what we have previously discussed. The code in EH mechanism that actually raises (`throw` statements) and handles exceptions (statements in `catch` blocks) seems to be the most fragile, in which it is less covered and more difficult to identify faults.

6.3 On the Usefulness of XaviEH

Our empirical study was powered by XaviEH, a tool that automatically generates a complete analysis and report of EH coverage and mutation testing for a certain Java system. XaviEH can be easily employed by developers as an EH testing diagnostics tool. Based on XaviEH outputs, developers can plan and improve their test suites regarding EH code.

Furthermore, given its full automated features, XaviEH could be also accommodated in continuous integration pipelines. In this context, developers would receive EH testing reports in each commit, which could create and foster a culture of continuous improvement of EH testing practices. In addition, XaviEH's outputs could be employed as metrics and proxies of internal quality, as well as goals to be achieved by the development team.

7 THREATS TO THE VALIDITY

The threats to the validity of our investigation are discussed using the four threats classification (conclusion, construct, internal, and external validity) presented by (WOHLIN *et al.*, 2012).

7.1 Conclusion Validity

Threats to the conclusion validity are concerned with issues that affect the ability to draw correct conclusions regarding the treatment and the outcome of an experiment. To deal with this threat, we carefully chose proper statistical tests (KS, MW, Friedman, and Nemenyi tests) that have been investigated and validated in previous studies (KUMAR *et al.*, 2011), (JI *et al.*, 2009). We also selected correlation measures (Spearman's rank-order correlation coefficient) to investigate the relationship between different aspects of EH testing (by means of code coverage and mutation analysis) and its effectiveness. Additionally, we have observed the assumptions (e.g., samples distribution, dependence, and size) of all statistical tests we used, trying to avoid wrong conclusions. Finally, regarding the limited set of long-lived Java libraries, we collected them from open source communities following a carefully defined set of criteria to ensure the disposal of other libraries that were not aligned with the study.

7.2 Internal Validity

Threats to the internal validity are influences that can affect the independent variable with respect to causality, without the researcher's knowledge. Thus they threat the conclusion about a possible causal relationship between treatment and outcome. Even not being interested in drawing causal relationships in our study, we have identified that some independent variables not known by us have some influence on the relationship between the EH code coverage and the mutation scores distribution in the studied libraries.

7.3 Construct Validity

Construct validity concerns generalizing the result of the study to the concept or theory behind the study. To avoid inconsistencies in the interpretation of the results and research question, a peer debriefing approach was adopted for both research design validation and

document review. Additionally, we developed a tool, XaviEH, in order to automate most of the study's parts, with an aim to avoid or alleviate the occurrence of human-made mistakes (or bias) during the execution of our experiments.

7.4 External Validity

Threats to external validity are conditions that limit our ability to generalize the results of our study to industrial practice. The main threats to this validity are related to the domain and sample size (i.e., the 27 libraries) we used in this study. Concerning the sample domain, we try to deal with this threat by arguing that the library domain is an interesting one that presents several and different usage scenarios, which is quite interesting from the a testing evaluation point of view. Additionally, concerning the sample size, we dealt with this threat by using diversity and longevity criteria. We chose libraries from Apache and picked up other well-know libraries developed by other development teams to get more diversity in terms of team knowledge, skills, and coding practices. Finally, we chose libraries that are long-lived as a way to guarantee a degree of maturity and stability.

8 CONCLUSION AND FINAL REMARKS

In this study, we empirically explored EH testing practices by analyzing in which degree the EH code is covered by unit-test suites of 27 long-lived java libraries and how effective are these test suites in detecting artificially injected EH faults. Our findings suggest that, indeed, EH code is, in general, less covered than non-EH code. Additionally, we gather evidence indicating that the code within `catch` blocks and the `throw` statements have a low coverage degree. However, even being less covered, the mutation analysis shows that, the test suites are able to detect most of artificial EH faults.

To the best of our knowledge, this is the first study that empirically addresses this concern. Thus, the results achieved in this study can be seen as a starting point for further investigation regarding testing practices for EH code.

This study was deeply supported by the XaviEH tool. Without this level of automation, it would not be possible to manually extract and synthesize information regarding EH code coverage and EH mutation scores. Therefore, we freely turn it available to the community (LIMA *et al.*, 2020). We include XaviEH's source code and usage instructions.

As future work, we are interested in (i) investigating the performance of the libraries test suites against real-world bugs; (ii) investigating the performance of the libraries test suites in an software evolution scenario; (iii) exploring software systems from different domains; and (iv) inspecting the test suites to identify and catalog what practices make a test suite better than others regarding EH testing.

BIBLIOGRAPHY

AHMED, I.; GOPINATH, R.; BRINDESCU, C.; GROCE, A.; JENSEN, C. Can testedness be effectively measured? In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2016. (FSE 2016), p. 547–558. ISBN 978-1-4503-4218-6. Disponível em: http://doi.acm.org/10.1145/2950290.2950324.

Antinyan, V.; Derehag, J.; Sandberg, A.; Staron, M. Mythical unit test coverage. **IEEE Software**, v. 35, n. 3, p. 73–79, May 2018.

BARBOSA, E. A.; GARCIA, A.; BARBOS, S. D. J. Categorizing faults in exception handling: A study of open source projects. In: **Software Engineering (SBES), 2014 Brazilian Symposium on**. [*S. l.*: *s. n.*], 2014. p. 11–20.

BAVOTA, G.; De Lucia, A.; Di Penta, M.; OLIVETO, R.; PALOMBA, F. An experimental investigation on the innate relationship between quality and refactoring. **Journal of Systems and Software**, Elsevier Ltd., v. 107, p. 1–14, sep 2015. ISSN 01641212.

BLOCH, J. **Effective Java (2Nd Edition) (The Java Series)**. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN 9780321356680.

CACHO, N.; BARBOSA, E. A.; ARAUJO, J.; PRANTO, F.; GARCIA, A.; CESAR, T.; SOARES, E.; CASSIO, A.; FILIPE, T.; GARCIA, I. How does exception handling behavior evolve? an exploratory study in java and c# applications. In: IEEE. **Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on**. [*S. l.*], 2014. p. 31–40.

CACHO, N.; CÉSAR, T.; FILIPE, T.; SOARES, E.; CASSIO, A.; SOUZA, R.; GARCIA, I.; BARBOSA, E. A.; GARCIA, A. Trading robustness for maintainability: An empirical study of evolving c# programs. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 584–595. ISBN 978-1-4503-2756-5. Disponível em: http://doi.acm.org/10.1145/2568225.2568308.

CHANG, B.-M.; CHOI, K. A review on exception analysis. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 77, n. C, p. 1–16, set. 2016. ISSN 0950-5849.

CHEN, H.; DOU, W.; JIANG, Y.; QIN, F. Understanding exception-related bugs in large-scale cloud systems. In: **Proceedings of the 34rd ACM/IEEE International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2019. (ASE 2019).

COELHO, R.; ALMEIDA, L.; GOUSIOS, G.; DEURSEN, A. V.; TREUDE, C. Exception handling bug hazards in android. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 22, n. 3, p. 1264–1304, jun. 2017. ISSN 1382-3256.

DEMsAR, J. Statistical comparisons of classifiers over multiple data sets. **J. Mach. Learn. Res.**, JMLR.org, v. 7, p. 1–30, dez. 2006. ISSN 1532-4435. Disponível em: http://dl.acm.org/citation.cfm?id=1248547.1248548.

Digkas, G.; Lungu, M.; Avgeriou, P.; Chatzigeorgiou, A.; Ampatzoglou, A. How do developers fix issues and pay back technical debt in the apache ecosystem? In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*: *s. n.*], 2018. p. 153–163.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 106, n. C, p. 82–101, ago. 2015. ISSN 0164-1212.

ECK, M.; PALOMBA, F.; CASTELLUCCIO, M.; BACCHELLI, A. Understanding flaky tests: The developer's perspective. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: ACM, 2019. (ESEC/FSE 2019), p. 830–840. ISBN 978-1-4503-5572-8. Disponível em: http://doi.acm.org/10.1145/3338906.3338945.

FILHO, J. L. M.; ROCHA, L.; ANDRADE, R.; BRITTO, R. Preventing erosion in exception handling design using static-architecture conformance checking. In: **Proceedings of the 11th European Conference on Software Architecture**. Cham: Springer International Publishing, 2017. (ECSA '17), p. 67–83. ISBN 978-3-319-65831-5.

FRIEDMAN, M. A comparison of alternative tests of significance for the problem of m rankings. **The Annals of Mathematical Statistics**, The Institute of Mathematical Statistics, v. 11, n. 1, p. 86–92, 03 1940.

GALLARDO, R.; HOMMEL, S.; KANNAN, S.; GORDON, J.; ZAKHOUR, S. B. **The Java Tutorial: A Short Course on the Basics**. 6th. ed. [*S. l.*]: Addison-Wesley Professional, 2014. 864 p. (Java Series). ISBN 0134034082.

GARCIA, A. F.; RUBIRA, C. M.; ROMANOVSKY, A.; XU, J. A comparative study of exception handling mechanisms for building dependable object-oriented software. **Journal of Systems and Software**, v. 59, n. 2, p. 197–222, 2001. ISSN 0164-1212.

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. **IEEE Transactions on software Engineering**, IEEE, n. 2, p. 156–173, 1975.

GOPINATH, R.; JENSEN, C.; GROCE, A. Code coverage for suite evaluation by developers. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 72–82. ISBN 9781450327565. Disponível em: https://doi.org/10.1145/2568225.2568278.

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. Disponível em: http://doi.acm.org/10.1145/3238147.3238183.

INOZEMTSEVA, L.; HOLMES, R. Coverage is not strongly correlated with test suite effectiveness. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 435–445. ISBN 978-1-4503-2756-5. Disponível em: http://doi.acm.org/10.1145/2568225.2568271.

INOZEMTSEVA, L.; HOLMES, R. Coverage is not strongly correlated with test suite effectiveness. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 435–445. ISBN 9781450327565. Disponível em: https://doi.org/10.1145/2568225.2568271.

IVANKOVIć, M.; PETROVIć, G.; JUST, R.; FRASER, G. Code coverage at google. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering**

Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM, 2019. (ESEC/FSE 2019), p. 955–963. ISBN 978-1-4503-5572-8. Disponível em: http://doi.acm.org/10.1145/3338906.3340459.

JI, C.; CHEN, Z.; XU, B.; WANG, Z. A new mutation analysis method for testing java exception handling. In: IEEE. **Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International**. [*S. l.*], 2009. v. 2, p. 556–561.

JUST, R.; JALALI, D.; INOZEMTSEVA, L.; ERNST, M. D.; HOLMES, R.; FRASER, G. Are mutants a valid substitute for real faults in software testing? In: **Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2014. (FSE 2014), p. 654–665. ISBN 978-1-4503-3056-5.

KECHAGIA, M.; SPINELLIS, D. Undocumented and unchecked: Exceptions that spell trouble. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. (MSR 2014), p. 312–315. ISBN 978-1-4503-2863-0.

KOCH, S. Software evolution in open source projects—a large-scale investigation. **J. Softw. Maint. Evol.**, John Wiley & Sons, Inc., USA, v. 19, n. 6, p. 361–382, nov. 2007. ISSN 1532-060X.

Kochhar, P. S.; Lo, D.; Lawall, J.; Nagappan, N. Code coverage and postrelease defects: A large-scale study on open source projects. **IEEE Transactions on Reliability**, v. 66, n. 4, p. 1213–1228, Dec 2017.

KOCHHAR, P. S.; THUNG, F.; LO, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [*S. l.*: *s. n.*], 2015. p. 560–564. ISSN 1534-5351.

KUMAR, K.; GUPTA, P.; PARJAPAT, R. New mutants generation for testing java programs. In: SPRINGER. **International Conference on Advances in Communication, Network, and Computing**. [*S. l.*], 2011. p. 290–294.

LIMA, L.; ROCHA, L.; BEZERRA, C.; PAIXAO, M. **Replication package for the paper: "Assessing Exception Handling Testing Practices in Open-Source Software Systems"**. 2020. We do not wish to make work under review publicly available. Nevertheless, we are happy to privately share with reviewers and editors if requested. Disponível em: https://to.be.disclosed.

MARINESCU, C. Are the classes that use exceptions defect prone? In: ACM. **Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution**. [*S. l.*], 2011. p. 56–60.

MARINESCU, C. Should we beware the exceptions? an empirical study on the eclipse project. In: IEEE. **Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on**. [*S. l.*], 2013. p. 250–257.

Martins, A. L.; Hanazumi, S.; Melo, A. C. V. d. Testing java exceptions: An instrumentation technique. In: **2014 IEEE 38th International Computer Software and Applications Conference Workshops**. [*S. l.*: *s. n.*], 2014. p. 626–631. ISSN null.

MCMINN, P. Search-based software test data generation: a survey. **Software testing, Verification and reliability**, Wiley Online Library, v. 14, n. 2, p. 105–156, 2004.

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013**. New York, New York, USA: ACM Press, 2013. p. 466.

OLIVEIRA, J.; BORGES, D.; SILVA, T.; CACHO, N.; CASTOR, F. Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 136, n. C, p. 1–18, fev. 2018. ISSN 0164-1212. Disponível em: https://doi.org/10.1016/j.jss.2017.10.032.

OSMAN, H.; CHIs, A.; CORRODI, C.; GHAFARI, M.; NIERSTRASZ, O. Exception evolution in long-lived java systems. In: **Proceedings of the 14th International Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press, 2017. (MSR '17), p. 302–311. ISBN 978-1-5386-1544-7.

PáDUA, G. B. de; SHANG, W. Revisiting exception handling practices with exception flow analysis. In: **2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [*S. l.*: *s. n.*], 2017. p. 11–20.

PáDUA, G. B. de; SHANG, W. Studying the prevalence of exception handling anti-patterns. In: **Proceedings of the 25th International Conference on Program Comprehension**. Piscataway, NJ, USA: IEEE Press, 2017. (ICPC'17), p. 328–331. ISBN 978-1-5386-0535-6.

PáDUA, G. B. de; SHANG, W. Studying the relationship between exception handling practices and post-release defects. In: **Proceedings of the 15th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2018. (MSR '18), p. 564–575. ISBN 978-1-4503-5716-6. Disponível em: http://doi.acm.org/10.1145/3196398.3196435.

PAIXAO, M.; KRINKE, J.; HAN, D.; RAGKHITWETSAGUL, C.; HARMAN, M. Are developers aware of the architectural impact of their changes? In: **ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering**. [*S. l.*: *s. n.*], 2017.

PAPADAKIS, M.; SHIN, D.; YOO, S.; BAE, D.-H. Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 537–548. ISBN 978-1-4503-5638-1. Disponível em: http://doi.acm.org/10.1145/3180155.3180183.

PAWLAK, R.; MONPERRUS, M.; PETITPREZ, N.; NOGUERA, C.; SEINTURIER, L. Spoon: A library for implementing analyses and transformations of java source code. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 46, n. 9, p. 1155–1179, set. 2016. ISSN 0038-0644. Disponível em: https://doi.org/10.1002/spe.2346.

RASHKOVITS, R.; LAVY, I. Students' misconceptions of java exceptions. In: RAHMAN, H.; MESQUITA, A.; RAMOS, I.; PERNICI, B. (Ed.). **Proceedings of the 7th Mediterranean Conference on Information Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. (MCIS '12), p. 1–21. ISBN 978-3-642-33244-9.

Reales, P.; Polo, M.; Fernández-Alemán, J. L.; Toval, A.; Piattini, M. Mutation testing. **IEEE Software**, v. 31, n. 3, p. 30–35, May 2014. ISSN 1937-4194.

ROMANO, D.; PENTA, M. D.; ANTONIOL, G. An approach for search based testing of null pointer exceptions. In: IEEE. **2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**. [*S. l.*], 2011. p. 160–169.

SAWADPONG, P.; ALLEN, E. B. Software defect prediction using exception handling call graphs: A case study. In: IEEE. **High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on**. [*S. l.*], 2016. p. 55–62.

SCHWARTZ, A.; PUCKETT, D.; MENG, Y.; GAY, G. Investigating faults missed by test suites achieving high code coverage. **Journal of Systems and Software**, v. 144, p. 106 – 120, 2018. ISSN 0164-1212.

SHAH, H.; GöRG, C.; HARROLD, M. J. Why do developers neglect exception handling? In: **Proceedings of the 4th International Workshop on Exception Handling**. New York, NY, USA: ACM, 2008. (WEH '08), p. 62–68. ISBN 978-1-60558-229-0.

SHAH, H.; GORG, C.; HARROLD, M. J. Understanding exception handling: Viewpoints of novices and experts. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 36, n. 2, p. 150–161, mar. 2010. ISSN 0098-5589.

SHAH, H.; HARROLD, M. J. Exception handling negligence due to intra-individual goal conflicts. In: **Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (CHASE '09), p. 80–83. ISBN 978-1-4244-3712-2.

SHAHROKNI, A.; FELDT, R. A systematic review of software robustness. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 1, p. 1–17, jan. 2013. ISSN 0950-5849.

SHI, L.; ZHONG, H.; XIE, T.; LI, M. An empirical study on evolution of api documentation. In: **Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software**. Berlin, Heidelberg: Springer-Verlag, 2011. (FASE'11/ETAPS'11), p. 416–431. ISBN 978-3-642-19810-6.

SINHA, S.; HARROLD, M. J. Analysis and testing of programs with exception handling constructs. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 26, n. 9, p. 849–871, set. 2000. ISSN 0098-5589. Disponível em: http://dx.doi.org/10.1109/32.877846.

VIEIRA, R.; SILVA, A. da; ROCHA, L.; GOMES, J. a. P. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. In: **Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2019. (PROMISE'19), p. 80–89. ISBN 9781450372336. Disponível em: https://doi.org/10.1145/3345629.3345639.

WIRFS-BROCK, R. J. Toward exception-handling best practices and patterns. **IEEE Software**, v. 23, n. 5, p. 11–13, Sept 2006. ISSN 0740-7459.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [*S. l.*]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

YANG, Y.; ZHOU, Y.; SUN, H.; SU, Z.; ZUO, Z.; XU, L.; XU, B. Hunting for bugs in code coverage tools via randomized differential testing. In: **Proceedings of the 41st International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2019. (ICSE '19), p. 488–499. Disponível em: https://doi.org/10.1109/ICSE.2019.00061.

ZHAI, H.; CASALNUOVO, C.; DEVANBU, P. Test coverage in python programs. In: **Proceedings of the 16th International Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press, 2019. (MSR '19), p. 116–120. Disponível em: https://doi.org/10.1109/MSR.2019.00027.

ZHANG, P.; ELBAUM, S. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 32:1–32:28, set. 2014. ISSN 1049-331X. Disponível em: http://doi.acm.org/10.1145/2652483.

Zhong, H.; Mei, H. An empirical study on api usages. **IEEE Transactions on Software Engineering**, v. 45, n. 4, p. 319–334, April 2019.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 29, n. 4, p. 366–427, dez. 1997. ISSN 0360-0300. Disponível em: http://doi.acm.org/10.1145/267580.267590.