Empirical Software Engineering (2021) 26: 85
https://doi.org/10.1007/s10664-021-09983-3

Assessing exception handling testing practices in open-source libraries

Luan P. Lima[1] · Lincoln S. Rocha[1] · Carla I. M. Bezerra[2] · Matheus Paixao[3]

Accepted: 26 May 2021 / Published online: 22 June 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

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 exception handling (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 ecosystems. 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 libraries presented test suites capable of detecting more than 70% of the injected faults. From a total of 12,331 mutants created in this study, the test suites were able to detect 68% of them.

Keywords Exception handling testing · Mutation analysis · Adequacy measurement · Effectiveness measurement · Exploratory study

1 Introduction

Exception handling (EH) is a forward-error recovery technique used to improve software robustness (Shahrokni and 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

Communicated by: Alexander Serebrenik

✉ Lincoln S. Rocha
lincoln@dc.ufc.br

Extended author information available on the last page of the article.

(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 and Allen 2016). Such studies have shown that the effectiveness of EH code is directly linked to the overall software quality (de Pádua and Shang 2017b; 2018). To ensure and assess the EH code, developers make use of software testing, which, in this context, is referred to as EH testing (Sinha and Harrold 2000; Martins et al. 2014; Zhang and Elbaum 2014).

Despite the importance and the existence of usage patterns and guidelines for EH implementation (Wirfs-Brock 2006; Bloch 2008; Gallardo et al. 2014; Jenkov 2013), 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 and Harrold 2009; Shah et al. 2010; Rashkovits and Lavy 2012; Kechagia and Spinellis 2014; Chang and Choi 2016; Dalton et al. 2020). 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 effectiveness.

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 (Hunt and Thomas 2003; Gulati and Sharma 2017) and rules of thumb.[1]

Nevertheless, to evaluate 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 and 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 effectiveness (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. 2019). 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 libraries. We developed a tool, called XaviEH, to assist in these analyses. XaviEH employs both coverage and mutation analysis as proxies for EH testing effectiveness in a certain system. In addition, XaviEH uses tailored

[1] http://wiki.c2.com/?ExceptionPatterns

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 assess whether there are types of EH bugs that are more difficult to detect by the studied libraries' test suites than others. Finally, based on the analysis by XaviEH, we ranked these libraries to assess which ones present significantly better indicators of EH testing effectiveness.

The main contributions of this paper are listed as follows:

- The first empirical study to evaluate adequacy and effectiveness of EH testing practices in open-source libraries.
- 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. 2021).

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

2 Background

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

2.1 Software Test Criteria and Adequacy

Goodenough and 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: statement coverage, branch coverage, and function/method coverage (Antinyan et al. 2018). 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 paper, 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. According to Ammann and Offutt (2016), mutation testing is commonly used as a "gold standard" in experimental studies for comparative evaluation of other test criteria. 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 (1), indicates the ratio of killed mutants compared to all created mutants (Zhu et al. 1997; Ammann and Offutt 2016). Mutation score indicates the effectiveness of a certain test suite, as it evaluates the test suite's ability to find defects.

$$MutationScore(s) = \frac{|KM(s)|}{|M(s)|} \tag{1}$$

There are cases where it is not possible to find a test case that could kill a mutant. The mutant is behaviorally equivalent to the original program. This kind of mutants are referred to as equivalent mutants. Therefore, to obtain a more accurate mutation score, it is necessary remove the equivalent mutants $E(s)$ from the set of created mutants, resulting in improved definition of mutation score, as depicted in (2).

$$MutationScore(s) = \frac{|KM(s)|}{|M(s) - E(s)|} \tag{2}$$

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 3.2).

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 inherit, directly or indirectly, from Java's `Exception` class, except those that inherit, directly or indirectly, from `Error` or `RuntimeException` classes, named unchecked ones. Checked exceptions represent exceptional conditions that, hypothetically, a robust software should be able to recover from. Unchecked exceptions represent an internal (`RuntimeException`) or an external (`Error`) exceptional condition 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, with or without an exception occurring and/or being handled. Finally blocks are commonly used for coding cleanup actions.

3 Experimental Design

Our study aims at investigating practices for EH testing in open-source libraries. To achieve this, we selected 27 long-lived Java libraries to serve as subjects in our empirical evaluation (see Section 3.1). 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 Section 3.4) 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 libraries. 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 (Demšar 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 section 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 are available in our replication package (Lima et al. 2021).

3.1 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; de Pádua and Shang 2018; Chen et al. 2019). Second, EH testing is considered more challenging than non-EH testing due to its (i) complex runtime and (ii) *flakiness* (Zhang and Elbaum 2014; Eck et al. 2019). Flaky tests are the ones that can intermittently pass or fail even for the same code version (Luo et al. 2014). Consider a test aimed at reproducing the (un)availability of resources at runtime, such as internet connection or databases, for example. Most of such resources cannot be easily mocked, and the test execution is bound to an external state of the system, which may cause a flaky behavior.

Hence, to properly study EH testing, we need long-lived subject systems that cater 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 and testing.

Therefore, we turned our attention to the Apache Software Foundation ecosystem.[2] 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 years 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 and 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.[3] 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 Gradle 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 tools we used to build XaviEH (see Section 3.2). To identify long-lived libraries, we computed the distribution of the age of all Apache Commons's libraries in years. Hence, we considered long-lived systems to be all libraries above the 3rd quartile

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

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

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 section of Table 1. Nevertheless, while fit 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 section of Table 1. In total, our empirical study considered 27 long-lived libraries from different open-source ecosystems.

Table 1 Summary of selected libraries. While the first section depicts the libraries from Apache Commons, the second section 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

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.

3.2 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 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. However, before explaining in detail the XaviEH execution steps, it is worth to mention the limitations of existing tools for test coverage regarding EH code.

Overall, existing test coverage tools (e.g., Cobertura[4], JaCoCo[5], and OpenClover[6]) compute all instruction, branch, and method coverages, and outputs such information within a coverage report following a specific format. In JaCoCo, for instance, one can choose to generate an XML-based report, which follows a well-defined DTD format[7]. Figure 1 shows an example fragment of a JaCoCo XML-based report[8]. On the upper part of Fig. 1, one can see the class name that was the target of the test coverage. In the middle of Fig. 1, for each line of code (comments and empty lines are not taken into account), the report gives the following information: the line number in the source code file (nr), the number of missed instructions (mi), covered instructions (ci), missed branches (mb), and covered branches (cb). Finally, at the bottom of Fig. 1, the report provides a summary for the class under consideration concerning the JaCoCo general coverage metrics (e.g., instruction, branch, line of code, and method). From this report, one may see that the main limitation of JaCoCo (which is shared by other test coverage tools) is that it does not differ EH code from non-EH code. Thus, this information remains hidden in the coverage report. To overcome this limitation, XaviEH employs static code analysis to determine which parts of the coverage report are related to EH code and non-EH code.

Figure 2 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[9] to download the source code. Otherwise, one can simply provide the local source code path 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`[10] and `gradle-tooling-api`[11] libraries to run the tests automatically and to identify whether a test suite is passing or not.

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

[4] http://cobertura.github.io/cobertura/
[5] https://www.jacoco.org/jacoco/
[6] http://openclover.org/
[7] https://www.jacoco.org/jacoco/trunk/coverage/report.dtd
[8] https://www.jacoco.org/jacoco/trunk/coverage/jacoco.xml
[9] https://www.eclipse.org/jgit/
[10] https://maven.apache.org/shared/maven-invoker/
[11] https://docs.gradle.org/current/userguide/embedding.html

```xml
<sourcefile name="ExecDumpClient.java">
  <line nr="38" mi="0" ci="2" mb="0" cb="0"/>
  <line nr="39" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="40" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="41" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="42" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="43" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="52" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="53" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="62" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="63" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="73" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="74" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="83" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="84" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="99" mi="0" ci="6" mb="0" cb="0"/>
  <line nr="115" mi="0" ci="4" mb="0" cb="0"/>
  <line nr="116" mi="0" ci="5" mb="0" cb="0"/>
  <line nr="118" mi="0" ci="6" mb="0" cb="0"/>
  <line nr="120" mi="0" ci="6" mb="0" cb="0"/>
  <line nr="122" mi="0" ci="4" mb="0" cb="0"/>
  <line nr="123" mi="0" ci="4" mb="0" cb="0"/>
  <line nr="126" mi="0" ci="6" mb="0" cb="0"/>
  <line nr="128" mi="0" ci="3" mb="0" cb="2"/>
  <line nr="129" mi="0" ci="5" mb="0" cb="0"/>
  <line nr="133" mi="0" ci="2" mb="0" cb="0"/>
  <line nr="135" mi="0" ci="2" mb="0" cb="0"/>
  <line nr="140" mi="0" ci="2" mb="0" cb="0"/>
  <line nr="143" mi="0" ci="4" mb="0" cb="0"/>
  <line nr="144" mi="0" ci="6" mb="0" cb="0"/>
  <line nr="145" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="146" mi="0" ci="5" mb="0" cb="2"/>
  <line nr="147" mi="0" ci="2" mb="0" cb="0"/>
  <line nr="149" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="150" mi="0" ci="2" mb="0" cb="0"/>
  <line nr="151" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="157" mi="0" ci="3" mb="0" cb="0"/>
  <line nr="158" mi="1" ci="0" mb="0" cb="0"/>
  <line nr="159" mi="4" ci="0" mb="0" cb="0"/>
  <line nr="160" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="161" mi="0" ci="1" mb="0" cb="0"/>
  <line nr="175" mi="1" ci="0" mb="0" cb="0"/>
  <line nr="186" mi="1" ci="0" mb="0" cb="0"/>
  <counter type="INSTRUCTION" missed="7" covered="115"/>
  <counter type="BRANCH" missed="0" covered="4"/>
  <counter type="LINE" missed="4" covered="38"/>
  <counter type="COMPLEXITY" missed="2" covered="11"/>
  <counter type="METHOD" missed="2" covered="9"/>
  <counter type="CLASS" missed="0" covered="1"/>
</sourcefile>
```


Fig. 1 Fragment of a JaCoCo XML-based test coverage report

operators we employ (see Section 3.3). By doing so, XaviEH creates an in-memory data structure that tracks eligible classes and mutation operators that can be applied to each class.

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

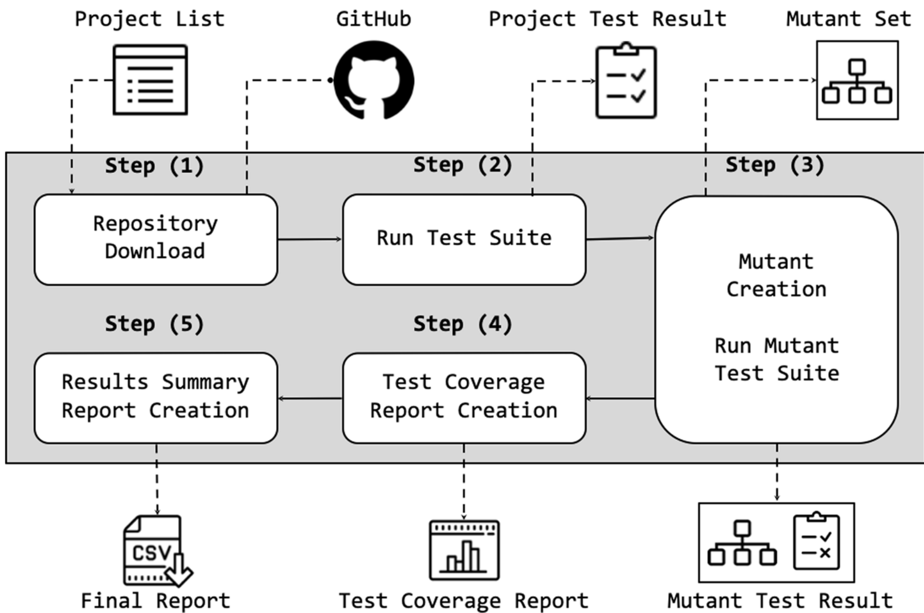

Fig. 2 Internal steps performed by XaviEH when evaluating the EH testing practices of a software system

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[12], 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 using JaCoCo, a Java code coverage library for monitoring and tracking code coverage. JaCoCo was chosen for this step for a couple of reasons. First, some Apache projects already employ JaCoCo as their official tool for test code analysis within their projects[13]. Second, JaCoCo is the tool of choice in previous related empirical studies (Saha et al. 2018; Turner et al. 2016). Hence, by employing a tool used by both practitioners and researchers, we would enhance XaviEH's relevance and actionability. Finally, one of the authors of the paper had previous experience with JaCoCo, which gave us more control over its integration into XaviEH.

Hence, XaviEH uses JaCoCo to generate a XML-based coverage report for each system under analysis. Next, for each system under consideration, XaviEH uses information from the coverage report and the static code analysis data provided by Spoon to determine what code parts referred to in the JaCoCo report are related or not to EH code. For instance, XaviEH employs Spoon to identify which lines of code of a class are within

[12]https://spoon.gforge.inria.fr/

[13]https://commons.apache.org/proper/commons-io/project-reports.html

try-catch-finally blocks and triangulates such information with code line numbers provided by JaCoCo report to track what instructions and branches within try-catch-finally blocks are covered or not. Finally, for each system under analysis, XaviEH collects all information needed to compute a suite of 24 test coverage metrics, as detailed in Section 3.4.

Finally, in Step (5), XaviEH summarizes the mutant and coverage analysis for the system under study. It generates two main reports in CSV files. The first one contains a pair of values (in the columns) needed to compute the coverage metrics. For instance, consider the metric CATCH_IC (see Section 3.4). In the report, we have the number of instructions missed (CATCH_MI) and covered (CATCH_CI) within the catch blocks. In this case, CATCH_IC is computed as CATCH_CI/(CATCH_MI + CATCH_CI). The second report file contains, for each mutation operator, the number of mutants killed and alive (in the columns), making it easy to compute the mutation score (see Section 2.2). It is important to notice that XaviEH does not perform any kind of statistical analysis and, as a limitation, it only can be employed in the analysis of programs written in Java, automatically built using Maven or Gradle, and that use JUnit to run its unit tests.

3.3 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).

The meaning of some mutation operators are very straightforward such as CBD, CRE, FBD, and TSD but the others are not so simple. Thus, to ease the understanding, we provide

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

Operator	Transformation in the Code
CBR	*Catch Block Replacement*. Replaces the catch block with exception types present in the invoking exception hierarchy (IEH) (Ji et al. 2009).
CBI	*Catch Block Insertion*. Creates complete catch modules to conceal all types of exceptions (Ji et al. 2009).
CBD	*Catch Block Deletion*. Deletes the whole catch block to propagate the thrown exceptions (Ji et al. 2009).
PTL	*Placing Try Block Later*. Brings into the try block, statements placed after the try block that reference variables inside the try block (Ji et al. 2009).
CRE	*Catch and Rethrow Exception*. Re-throws the caught exceptions which are propagated to the upper modules (Ji et al. 2009).
FBD	*Finally Block Deletion*. Deletes the whole finally block to propagate the thrown exceptions (Kumar et al. 2011).
TSD	*Throw Statement Deletion*. Deletes the throw statement that should raise an exception (Kumar et al. 2011).

in Fig. 3 an illustrative exception handling code sample (1) and its two exception hierarchies (2) and (3). The semantic exception hierarchy (2) indicates the inheritance relationship between the exception types involved in the EH scenario. The invoking exception hierarchy (3) aims at organizing the structure of the program according to the relationship of different exception handlers using information from the method calls' chain. In (3), the exception type of the `catch` block attached to the `try` block in `methodOne()` (caller method) represents the root node (i.e., `FileNotFoundException`). Each type of exception (i.e., `IllegalArgumentException` and `IOException`) thrown by `methodTwo()`, which is called in the `try` block of `methodOne()`, is linked to the root node. These hierarchies help mutation operators determine how to transform the original program and inject defects.

Based on the EH code of Fig. 3, we provide in Fig. 4 an illustrative example of the transformations performed by each EH mutation operator.

3.4 Code Coverage Metrics

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

We considered four sets of coverage metrics. In the first set, we computed the overall instruction, branch and method coverage, i.e., considering the library's entire code base

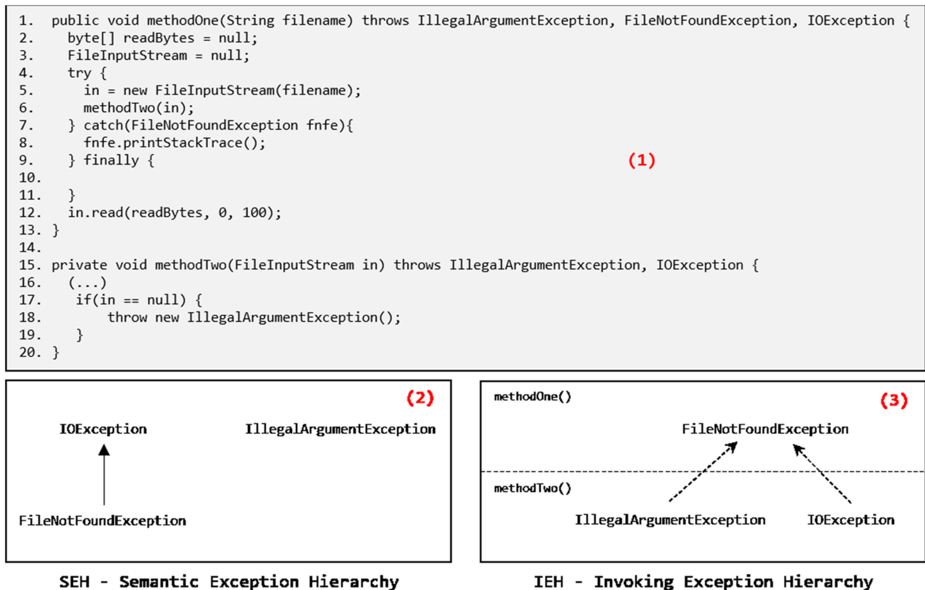

Fig. 3 An illustrative example of exception handling source code (1), semantic exception hierarchy (2), and invoking exception hierarchy (3) adapted from (Ji et al. 2009)

CBR
```
Original code:
7. }catch(FileNotFoundException fnfe) {...}

CBR mutant 1 (replace with one type ∈ IEH):
}catch(IOException ioe) {...}

CBR mutant 2 (replace with another type ∈ IEH):
}catch(IllegalArgumentException iae) {...}
```

CBI
```
Original code:
7. }catch(FileNotFoundException fnfe) {...}

CBI mutant (add all types ∈ IEH):
}catch(FileNotFoundException fnfe) {...}
catch(IOException ioe) {...}
catch(IllegalArgumentException iae) {...}
```

CBD
```
Original code:
7. }catch(FileNotFoundException fnfe) {...}

CBD mutant:
```

CRE
```
Original code:
7. }catch(FileNotFoundException fnfe) {
8.   fnfe.printStackTrace();
9. }

CRE mutant:
7. }catch(FileNotFoundException fnfe) {
8.   fnfe.printStackTrace();
9.   throw fnfe;
10.}
```

PTL
```
Original code:
3. FileInputStream = null;
4. try {
5.   in = new FileInputStream(filename);
6.   methodTwo(in);
7. } catch(FileNotFoundException fnfe){...}
    (...)
12. in.read(readBytes, 0, 100);

PTL mutant:
3. FileInputStream = null;
4. try {
5.   in = new FileInputStream(filename);
6.   methodTwo(in);
7.   in.read(readBytes, 0, 100);
8. } catch(FileNotFoundException fnfe){...}
    (...)
```

FBD
```
Original code:
7. } catch(FileNotFoundException fnfe){
8.   fnfe.printStackTrace();
9. } finally {
10.
11.}

FBD mutant:
7. } catch(FileNotFoundException fnfe){
8.   fnfe.printStackTrace();
9. }
```

TSD
```
Original code:
17. if(in == null) {
18.   throw new IllegalArgumentException();
19. }

TSD mutant:
17. if(in == null) {
18.
19. }
```

Fig. 4 An illustrative example of the transformations performed by each EH mutation operator regarding the EH code example shown in Fig. 3

(EH code **and** non-EH code). This is necessary for us to have a baseline of each library'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 section of Table 3. Next, the `TRY_CATCH_BC` coverage metric represents a specific EH-related metric that counts different `catch` blocks associated with a `try` block being selected depending on the raised exception type as a case of branching. It is important to notice that `BC` metric of the first section of Table 3 does not take `try-catch` statements into account to compute branch coverage, which make `TRY_CATCH_BC` and `BC` distinct coverage metrics.

Next, the coverage metrics in the third set are tailored for EH code. It considers the instructions and branches inside `try`, `catch` and `finally` blocks. These are detailed in the third section of Table 3. Finally, the fourth set of coverage metrics is applied to the code outside `try`, `catch` and `finally` blocks, i.e., non-EH code. Considering the overall coverage metrics as the baseline (first section), the non-EH coverage metrics (fourth section) are mathematical complements to the EH coverage metrics (third section). For instance, `BC = EH_BC + NON_EH_BC`.

Table 3 Code coverage metrics computed by XaviEH

Metric	Meaning
IC	*Instruction Coverage*. The percentage of instructions exercised by the test suite.
BC	*Branch Coverage*. The percentage of branches exercised by the test suite.
MC	*Method Coverage*. The percentage of methods exercised by the test suite.
TRY_CATCH_BC	*Try-Catch Branch Coverage*. The percentage of branches of try-catch exercised by the test suite. Each catch block associated with a try block can be seen as a possible branch based on the exception type.
EH_IC	*Exception Handling Instruction Coverage*. Instructions, catch, and finally blocks plus all throw instructions exercised by the test suite.
EH_BC	*Exception Handling Branch Coverage*. The percentage of branches in try, catch, and finally blocks exercised by the test suite.
TRY_IC	*Try Instruction Coverage*. The percentage of instructions in try blocks exercised by the test suite.
TRY_BC	*Try Branch Coverage*. The percentage of branches in try blocks exercised by the test suite.
CATCH_IC	*Catch Instruction Coverage*. The percentage of instructions in catch blocks exercised by the test suite.
CATCH_BC	*Catch Branch Coverage*. The percentage of branches in catch blocks exercised by the test suite.
FINALLY_IC	*Finally Instruction Coverage*. The percentage of instructions in finally blocks exercised by the test suite.
FINALLY_BC	*Finally Branch Coverage*. The percentage of branches in finally blocks exercised by the test suite.
THROW_IC	*Throw Instruction Coverage*. The percentage of throw instructions exercised by the test suite.
THROWS_MC	*Throws Method Coverage*. The percentage of methods with a throws clause in its signature exercised by the test suite.
NON_EH_IC	*Non-Exception Handling Instruction Coverage*. The percentage of instructions exercised by the test suite that are not throw and not in try, catch, and finally blocks.
NON_EH_BC	*Non-Exception Handling Branch Coverage*. The percentage of branches exercised by the test suite that are not in try, catch, and finally blocks.
NON_TRY_IC	*Non-Try Instruction Coverage*. The percentage of instructions exercised by the test suite that are not in try blocks.
NON_TRY_BC	*Non-Try Branch Coverage*. The percentage of branches exercised by the test suite that are not in try blocks.
NON_CATCH_IC	*Non-Catch Instruction Coverage*. The percentage of instructions exercised by the test suite that are not in catch blocks.
NON_CATCH_BC	*Non-Catch Branch Coverage*. The percentage of branches exercised by the test suite that are not in catch blocks.
NON_FINALLY_IC	*Non-Finally Instruction Coverage*. The percentage of instructions exercised by the test suite that are not in finally blocks.
NON_FINALLY_BC	*Non-Finally Branch Coverage*. The percentage of branches exercised by the test suite that are not in finally blocks.
NON_THROW_IC	*Non-Throw Instruction Coverage*. The percentage of instructions exercised by the test suite that are not throw.
NON_THROWS_MC	*Non-Throws Method Coverage*. The percentage of methods exercised by the test suite without a throws clause in its signature.

The first and second set of metrics are computed considering the library's entire code base. The third set of coverage metrics are specific for exception handling code. Finally, the fourth set is composed of mathematical complements to the exception handling coverage metrics in the third section

4 Study Results

4.1 Preliminary Observation of the Libraries' Overall Coverage

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

Figure 5 presents boxplots depicting the distribution of overall instruction, branch, and method coverage, as detailed in Section 3.4 and Table 3. Note that the distributions were computed considering all the 27 studied libraries. The median values for Instruction Coverage (`IC`), Branch Coverage (`BC`) and Method Coverage (`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 libraries in all metrics. This indicates that the libraries under study exhibit mature testing practices for the libraries' overall source code.

4.2 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 the computed code coverage metrics of the first three sections of Table 3 for all libraries included in this study.Not all metrics could be computed for all libraries. 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

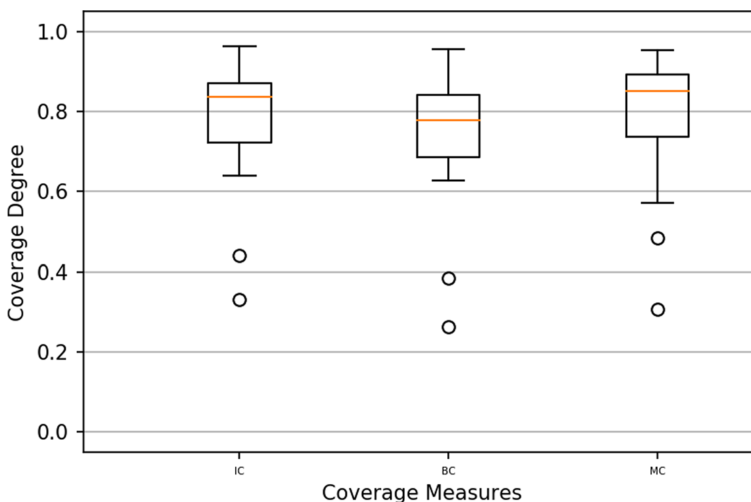

Fig. 5 Overall code coverage boxplots of the 27 studied libraries (`IC` - Instruction Coverage, `BC` - Branch Coverage, and `MC` - Method Coverage)

Table 4 Computed general code coverage measures per library

Library	IC	BC	MC	TRY_CATCH_BC	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	0.02	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	0.30	0.52	0.48	0.68	0.65	0.32	0.26	-	-	0.43	0.78
CLI	0.96	0.93	0.94	0.73	0.97	1.00	1.00	1.00	1.00	-	1.00	-	0.90	1.00
Codec	0.96	0.91	0.90	0.55	0.78	0.97	0.84	0.97	0.50	-	0.88	-	0.62	0.87
Collections	0.87	0.81	0.87	0.33	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	0.22	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	0.50	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	0.04	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	0.29	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	0.16	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	0.16	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	0.10	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	0.26	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	0.42	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	0.65	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	0.47	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	0.07	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	0.51	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	0.48	0.55	0.43	0.52	0.43	1.00	-	-	-	0.62	1.00
Validator	0.86	0.76	0.81	0.30	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	0.28	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	0.22	0.66	1.00	1.00	1.00	0.55	-	-	-	0.26	1.00
Jsoup	0.84	0.78	0.85	0.38	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	0.44	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	0.49	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	0.17	0.65	0.72	0.83	0.75	0.13	0.17	0.87	0.50	0.14	0.65

be computed. We indicate with a '-' all cases in which a certain coverage metric could not be computed for a certain library.

The boxplots in Fig. 6 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. 5 plus the `TRY_CATCH_BC` metric. The coverage degree of `EH_IC` ranges from 55% to 74%, `EH_BC` ranges from 54% to 78%, `THROWS_MC` ranges from 79% to 94%, and `TRY_CATCH_BC` ranges from 18% to 47%. Additionally, one should notice there exist libraries with 100% coverage for `EH_BC` and `THROWS_MC` and with about 2% for `TRY_CATCH_BC`.

We can draw interesting observations when comparing the EH-related coverage with the equivalent coverage metrics for the whole library displayed in Fig. 5. 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 libraries 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 libraries reached 100% EH method coverage, which did not occur for overall method coverage in any library. Particularly, looking at the `TRY_CATCH_BC` boxplot, we can also observe that coverage of `catch` blocks (i.e., the reachability of these blocks instead of the instructions or branches within them) is very low, indicating that the test suites of the studied libraries are not able to exercise the code derived from exceptional control flows.

We also plotted boxplots detailing the internal distribution of `EH_IC` (see Fig. 7) and `EH_BC` (see Fig. 8). Looking at Fig. 7 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` instructions and `catch` blocks have the worst coverage, with 25% and

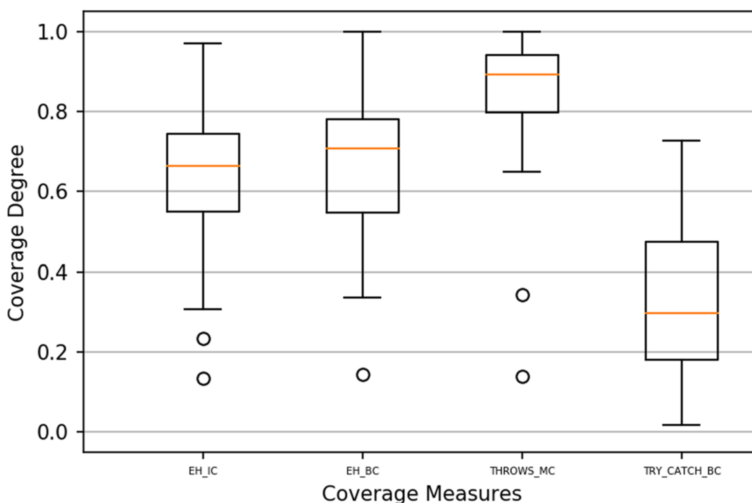

Fig. 6 Distribution of general EH-related code coverage metrics for the libraries under study

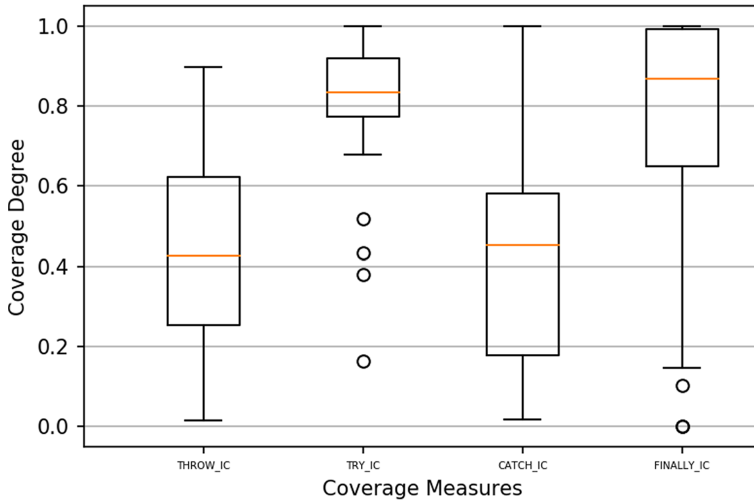

Fig. 7 EH instruction code coverage boxplots of studied libraries

17%, respectively. Hence, this suggests that THROW_IC and CATCH_IC are the metrics that impact the general EH_IC the most.

Looking at Fig. 8 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

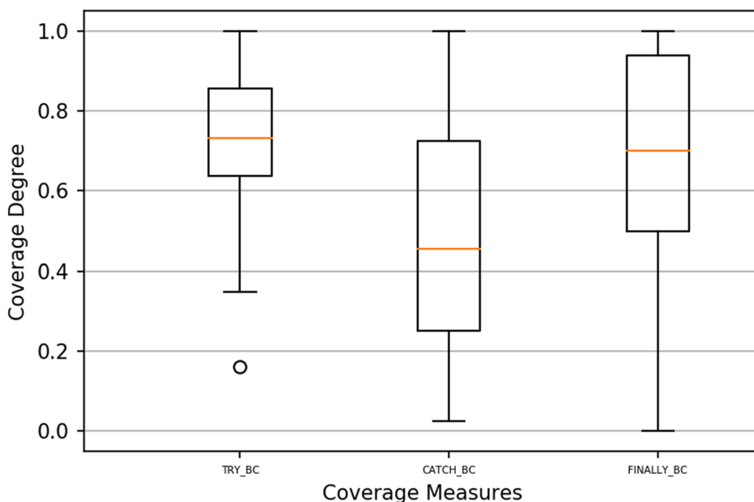

Fig. 8 EH branch code coverage boxplots of studied libraries

`throw` instructions and `catch` blocks. This is a worrisome observation because `try` and `finally` blocks are always executed in non-exceptional behaviors. Hence, we deduct that the test suites of the studied libraries are failing to raise internal (coded in the library) and external (signaled from third-party libraries as a return of a method call) 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 exceptional control flows are not being well tested. This is supported by the results of the `TRY_CATCH_BC` metric.

4.3 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 verify 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 in the next paragraphs. We also compute the effect size using Cliff's delta statistic (Cliff 1993), a non-parametric measure that quantifies the amount of difference between two groups of observations (EH and non-EH code coverage). This measure can be seen as a complementary analysis for the corresponding hypothesis testing and p-values. We display the values for the Cliff's delta measure and their respective interpretation in the last column of Table 6.

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 $\mathcal{H}_0^A$: $\texttt{NON_EH_IC} = \texttt{EH_IC}$. 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 considereing all libraries. However, in case the KS null hypothesis is rejected, we assume the alternative hypothesis $\mathcal{H}_1^A$: $\texttt{NON_EH_IC} \neq \texttt{EH_IC}$, 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. It is important to observe that the MW test is only performed if the null hypothesis of KS is rejected. Consider the instruction coverage metric, for example. The MW test considers the null hypothesis $\mathcal{H}_0^A$: $\texttt{NON_EH_IC} > \texttt{EH_IC}$. 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 ($\mathcal{H}_1^A$: $\texttt{NON_EH_IC} < \texttt{EH_IC}$), 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 $\alpha < 0.05$.

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, the statistics test, and the Cliff's Delta effect size results

KS Hypothesis	p-value	MW Hypothesis	p-value	Effect size
$\mathcal{H}_0^A$: NON_EH_IC = EH_IC (✗)	$5.7 \times 10^{-4*\dagger}$	$\mathcal{H}_0^A$: NON_EH_IC > EH_IC (✓)	$2.8 \times 10^{-4*\dagger}$	0.61
$\mathcal{H}_1^A$: NON_EH_IC ≠ EH_IC		$\mathcal{H}_1^A$: NON_EH_IC < EH_IC		(large)
$\mathcal{H}_0^B$: NON_EH_BC = EH_BC (✗)	$5.3 \times 10^{-7*}\dagger$	$\mathcal{H}_0^B$: NON_EH_BC > EH_BC (✗)	$1.4 \times 10^{-6*\dagger}$	0.80
$\mathcal{H}_1^B$: NON_EH_BC ≠ EH_BC		$\mathcal{H}_1^B$: NON_EH_BC < EH_BC		(large)
$\mathcal{H}_0^C$: NON_THROWS_MC = THROWS_MC (✗)	$4.5 \times 10^{-2*}$	$\mathcal{H}_0^C$: NON_THROWS_MC > THROWS_MC (✗)	$4.5 \times 10^{-2*}$	0.30
$\mathcal{H}_1^C$: NON_THROWS_MC ≠ THROWS_MC		$\mathcal{H}_1^C$: NON_THROWS_MC < THROWS_MC		(small)
$\mathcal{H}_0^D$: NON_THROW_IC = THROW_IC (✗)	$3.1 \times 10^{-6*\dagger}$	$\mathcal{H}_0^D$: NON_THROW_IC > THROW_IC (✓)	$1.5 \times 10^{-6*\dagger}$	0.82
$\mathcal{H}_1^D$: NON_THROW_IC ≠ THROW_IC		$\mathcal{H}_1^D$: NON_THROW_IC < THROW_IC		(large)
$\mathcal{H}_0^E$: NON_TRY_IC = TRY_IC (✓)	$1.0^{\dagger}$	$\mathcal{H}_0^E$: NON_TRY_IC > TRY_IC	N/A	0.06
$\mathcal{H}_1^E$: NON_TRY_IC ≠ TRY_IC		$\mathcal{H}_1^E$: NON_TRY_IC < TRY_IC		(negligible)
$\mathcal{H}_0^F$: NON_CATCH_IC = CATCH_IC (✗)	$5.0 \times 10^{-6*\dagger}$	$\mathcal{H}_0^F$: NON_CATCH_IC > CATCH_IC (✓)	$8.8 \times 10^{-6*\dagger}$	0.74
$\mathcal{H}_1^F$: NON_CATCH_IC ≠ CATCH_IC		$\mathcal{H}_1^F$: NON_CATCH_IC < CATCH_IC		(large)
$\mathcal{H}_0^G$: NON_FINALLY_IC = FINALLY_IC (✓)	$4.6 \times 10^{-1\dagger}$	$\mathcal{H}_0^G$: NON_FINALLY_IC > FINALLY_IC	N/A	0.16
$\mathcal{H}_1^G$: NON_FINALLY_IC ≠ FINALLY_IC		$\mathcal{H}_1^G$: NON_FINALLY_IC < FINALLY_IC		(small)
$\mathcal{H}_0^H$: NON_TRY_BC = TRY_BC (✓)	$1.0^{\dagger}$	$\mathcal{H}_0^H$: NON_TRY_BC > TRY_BC	N/A	0.12
$\mathcal{H}_1^H$: NON_TRY_BC ≠ TRY_BC		$\mathcal{H}_1^H$: NON_TRY_BC < TRY_BC		(negligible)
$\mathcal{H}_0^I$: NON_CATCH_BC = CATCH_BC (✗)	$5.4 \times 10^{-3*\dagger}$	$\mathcal{H}_0^I$: NON_CATCH_BC > CATCH_BC (✓)	$1.4 \times 10^{-3*\dagger}$	0.59
$\mathcal{H}_1^I$: NON_CATCH_BC ≠ CATCH_BC		$\mathcal{H}_1^I$: NON_CATCH_BC < CATCH_BC		(large)
$\mathcal{H}_0^J$: NON_FINALLY_BC = FINALLY_BC (✓)	$4.4 \times 10^{-1\dagger}$	$\mathcal{H}_0^J$: NON_FINALLY_BC > FINALLY_BC	N/A	0.14
$\mathcal{H}_1^J$: NON_FINALLY_BC ≠ FINALLY_BC		$\mathcal{H}_1^J$: NON_FINALLY_BC < FINALLY_BC		(negligible)

The symbols ✓ and ✗ indicate the result of the null hypothesis test (✓ fail to reject, and ✗ reject). The Cliff's Delta effect size interpretation: negligible = [0, 0.147), small = [0.147, 0.33), medium = [0.33, 0.474), and large = [0.474, 1]. The symbol $*$ indicates the p-value adjustment applying the Benjamini and Yekutieli (2001)'s method on the pairs of p-values obtained by the two consecutive statistical tests, KS and MW, per line. The symbol † indicates the p-value adjustment applying the (Benjamini and Yekutieli 2001)'s method on the sets of p-values where the metrics involved in each the test (per column) are not independent

To enhance this analysis, in Fig. 9, 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 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 $\mathcal{H}_0^A$: NON_EH_IC = EH_IC and did not reject the MW null hypothesis $\mathcal{H}_0^A$: 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 $\mathcal{H}_0^B$: NON_EH_BC = EH_BC and also reject the MW null hypothesis $\mathcal{H}_0^B$: NON_EH_BC > EH_BC. This is a counterintuitive observation given the results previously observed in our study. We address this during our study' discussion (see Section 5).

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 $\mathcal{H}_0^C$: NON_THROWS_MC = THROWS_MC and the MW null hypothesis $\mathcal{H}_0^C$: 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 section.

Figure 10 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 $\mathcal{H}_0^D$: NON_THROW_IC = THROW_IC and accept the MW null hypothesis $\mathcal{H}_0^D$: NON_THROW_IC > THROW_IC. This indicates that even though the

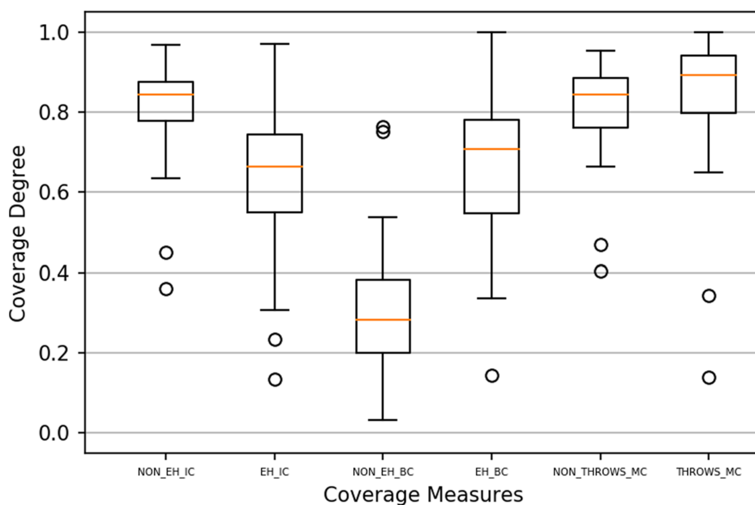

Fig. 9 Overall EH and non-EH code coverage boxplots of studied libraries

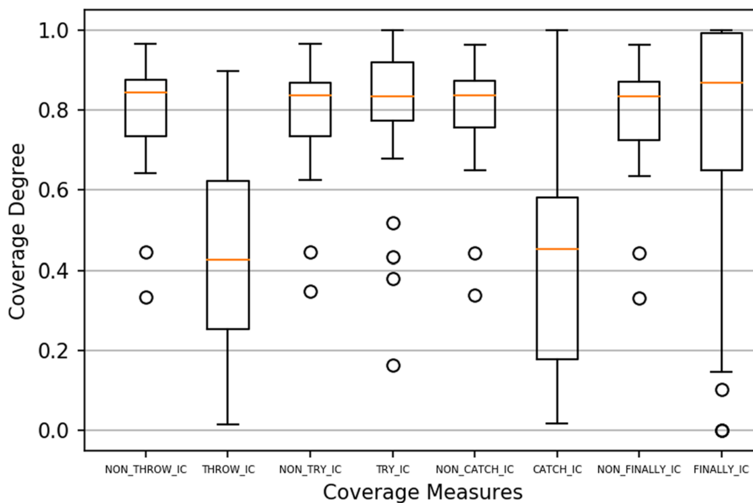

Fig. 10 The EH and non-EH instruction coverage boxplots of studied libraries

libraries under study present a fairly high level of instruction coverage (see Fig. 5), 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 $\mathcal{H}_0^F$: NON_CATCH_IC = CATCH_IC and accepting the MW null hypothesis $\mathcal{H}_0^F$: 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 ($\mathcal{H}_0^E$: NON_TRY_IC = TRY_IC and $\mathcal{H}_0^G$: 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. 11 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 $\mathcal{H}_0^I$: NON_CATCH_BC = CATCH_BC and accepting the MW null hypothesis $\mathcal{H}_0^I$: 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 realize they are similar. This perception is also confirmed by the statistical tests when both the KS and MW null hypotheses ($\mathcal{H}_0^H$: NON_TRY_BC = TRY_BC and $\mathcal{H}_0^J$: 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.

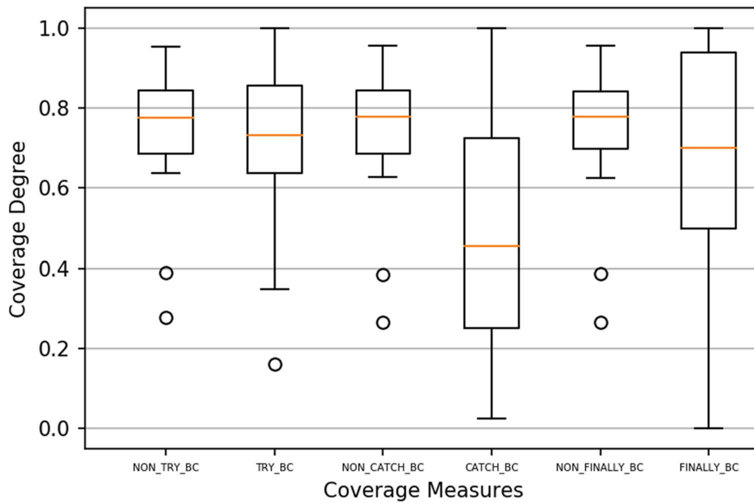

Fig. 11 The EH and non-EH branch coverage boxplots of studied libraries

4.4 RQ3. What is the Effectiveness of EH Testing in Long-Lived Java Libraries?

Summary of RQ3: The libraries under study present effective test suites for EH code, where 68% of all the injected defects are identified. However, the libraries 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 3.2. In Table 7, we present results of the mutation testing analysis we performed. For each mutation operator (see Section 3.3), we show the number of mutants killed by the test suite, the number of mutants left alive, and the mutation score.

Furthermore, in Fig. 12, we present boxplots showing the mutation score distribution for all libraries and each mutation operator. Note that we computed the distributions considering only the mutation scores of libraries in which we were able to generate at least 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`). Considering all 12,331 created mutants, we computed a global mutation score of 0.68, which means that 68% of all mutants were killed. Since we do not eliminate the equivalent mutants, this 0.68 can be seen as a lower boundary value for the mutation score.

When looking at both the table and figure, one must notice that the libraries under study achieved mean and median mutation scores 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 and Holmes 2014a). 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 libraries under study

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

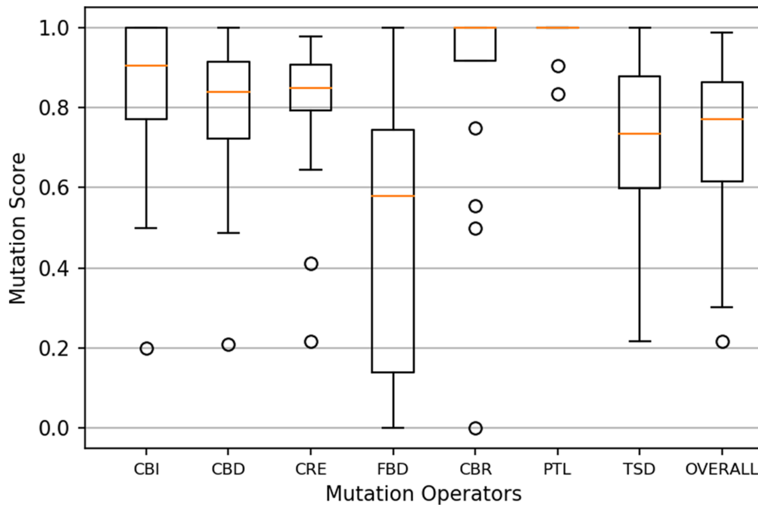

Fig. 12 Mutation scores distribution bloxpots

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 libraries. 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 libraries and `CBR` generating only 80 mutants overall. This is due to how each operator generates mutants (see Section 3.3). While `TSD` simply deletes a `throw` statement, `CBR` identifies a `catch` block and searches the exception hierarchy to replace the exception for a derived type. Hence, it is expected that there will be much more `throw` statements to be deleted throughout all libraries than derived exceptions in `catch` blocks to be replaced. However, there seems to be no relationship between the number of mutants generated to 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.

4.5 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 (1940), which aims at quantifying the consistency of the results obtained by a test suite when applied over several types of EH bugs. We applied each of the 7 mutation operators to each library, collecting the mutation score for each case. Next, we rank the 7 operators for each library according to their mutation score (the highest mutation score getting rank 1, the second-highest rank 2 and so on). We leverage the Friedman test to check whether the mutation score for any of the 7 mutation operators ranks consistently higher or lower than the others. 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 (Demšar 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 because Friedman's test guidelines state that p-values are reliable for more than 6 measurements[14] (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 `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 (3), 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 $q_{0.05} = 2.948$.

$$\mathrm{CD} = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \tag{3}$$

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

[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.friedmanchisquare.html

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**

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.

5 Discussion

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

5.1 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 libraries 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 these test suites do not properly cover EH code (statements in `catch` blocks) but also that these suites are not able to exercise the code parts responsible for raising exceptions (`throw` statements). We followed on this insight by computing two sets of Spearman's correlations considering all the libraries under study.

The first correlation was computed between `throw` instruction coverage (`THROW_IC`) and `catch` blocks' instruction coverage (`CATCH_IC`), and the second between `throw` instruction coverage (`THROW_IC`) and `catch` blocks branch (`CATCH_BC`) coverage. The results show a strong correlation in both cases with $\rho = 0.582664$ (`THROW_IC` and `CATCH_IC`) and $\rho = 0.674882$ (`THROW_IC` and `CATCH_BC`). The correlation results confirm that these coverage values are strongly connected, as empirically observed. We suggest two possible scenarios that may explain such correlations. In the first scenario, the `catch`

blocks not being covered are the ones responsible for catching the exceptions not being raised. Differently, in the second scenario, the two elements (throw statements and catch blocks) are not connected, which may include JRE and other external exceptions not being caught, for instance. Further studies are needed to thoroughly understand this phenomenon.

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 generate test cases according to a certain objective function automatically. 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 existing work in this direction (Romano et al. 2011).

5.2 On the Effectiveness of EH Testing

RQ3-4 shows 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 libraries presented mutation scores of more than 68% 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.

5.3 On Qualitatively Assessing our Results

Despite our paper having the main goal of quantitatively assessing EH testing practices in real-world libraries, we found a few interesting cases among the data in which a closer inspection may yield interesting insights.

First, we noticed that the coverage results for the different libraries presented a wide variation. For this analysis, we take the value of TRY_CATCH_BC as our metric of comparison. We chose this metric because it is a good representative of EH testing adequacy, as previously discussed in our results sections. Based on this metric, CLI presented the best coverage results with 73% of its catch blocks being covered. As its counterpart, BCEL only covers 2% of its catch blocks. This may be explained by the size of each library. While BCEL is composed of 344 classes and contains 143 catch blocks, CLI is composed of 21 classes and contains only 11 catch blocks. However, this behavior is not repeated through the whole dataset. Math, for instance, is the largest library with 740 classes, and it covers 47% of its 180 catch blocks. We suggest that further studies are needed to fully understand this phenomenon.

In our study, we collected data from Apache and non-Apache libraries. Although the number of libraries in both subsets does not allow for statistical comparisons, we can still observe a few interesting details. All of the non-Apache libraries achieve less than 50% catch blocks coverage, as according to TRY_CATCH_BC. Differently, 5 out of 21 Apache libraries cover more than half of its catch blocks. This observation is especially interesting because 3 of the non-Apache libraries are testing-related frameworks (Hamcrest, JUnit and Mockito). This attests to the well-known quality control of the Apache ecosystem.

5.4 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 testing effectiveness, as well as goals to be achieved by the development team.

6 Threats to 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).

6.1 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.

6.2 Internal Validity

Threats to 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.

6.3 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.

6.4 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 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.

7 Related Work

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

7.1 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; 2013). This correlation emerges from sub-optimal exception handling practices (i.e., anti-patterns and flow characteristics) current adopted by software developers (Sawadpong and Allen 2016; de Pádua and 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 and Elbaum 2014; Chang and 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 and 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.

de Pádua and Shang (2017a, b, 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, de and 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 might lead to software failures, their identification highlight the opportunities of leveraging automated software analysis to assist in exception handling practices.

de Pádua and 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 suggest 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 mentioned studies, our study investigates EH testing practices in 27 long-lived Java libraries with more than 11 years of active development. We generated a total of 12,331 software mutants and observed that the systems present effective test suites for EH code, where more than 68% of the defects were identified. However, the libraries present difficulties in identifying defects in `finally` blocks.

7.2 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 `catch` blocks, add statements to re-throws a caught exception, and try to rearrange `try` blocks by including statements with some relevant references after the `catch` blocks. In our study, we employed 7 mutation operators, 5 (`CBR`, `CBI`, `CBD`, `PTL`, and `CRE`) from Ji et al. (2009) and 2 (`FCD` we call `FBD` and `TSD`) from Kumar et al. (2011).

Zhang and 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.

Goffi et al. (2016) presented a technique to automatically generate test oracles for exceptional behavior, called Toradocu. Toradocu uses natural language processing to automatically extract conditional expressions regarding exceptional behavior from Javadoc comments. An empirical evaluation shows that Toradocu improves the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and reduces EvoSuite's false positives by 33%.

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 the 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 a varying impact 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.

Dalton et al. (2020) performed a study to understand how 417 open source Java projects are testing the exceptional behavior. They looked at test suites coded using JUnit and TestNG frameworks, and the AssertJ library. Overall, they count test methods that expect exceptions to be raised, which they called "exceptional behavior testing". They found that (i) 60.91% of projects have at least one test method dedicated to testing the exceptional behavior; (ii) the number of test methods for exceptional behavior with respect to the total number of test methods lies between 0% and 10% in 76.02% of projects; and (iii) 57.31% of projects test only up to 10% of the used exceptions in the system under test. They triangulate such results with a survey with 66 developers from the studied projects. The survey respondents confirm the findings and support the claim that developers often neglect exceptional behavior tests. However, different from our study, Dalton et al. (2020) did not evaluate the adequacy and effectiveness of exceptional behavior testings itself (i.e., did not run the test methods that exercise the exceptional behaviors of the system under test). Instead, they performed a static analysis of the testing code to gather evidence on whether or not there are test methods intended to test the exceptional behavior.

Our study used long-lived Java libraries and our results shows that in spite of the low coverage for instruction and branches related to EH code, the unit tests of the studied libraries were able to detect a significant amount of artificially injected faults.

7.3 Code Coverage

Gligoric et al. (2013) presented an extensive study that evaluates coverage criteria over non-adequate test suites. The authors analyzed a large set of plausible criteria, including statement and branch coverage, as well as stronger criteria used in recent studies. Two criteria performed best: branch coverage and intra-procedural acyclic path coverage. The study's results suggest that researchers should use branch coverage to compare suites whenever possible, but all evaluated criteria performed well to predict mutation scores.

Inozemtseva and Holmes (2014b) 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.

Chekam et al. (2017) conducted a study using a robust experimental methodology that provided evidence to support the claim that strong mutation testing yields high fault revelation, while statement, branch and weak mutation testing enjoy no such fault revealing ability. The findings also revealed that only the highest levels of strong mutation coverage attainment have strong fault-revealing potential.

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) argues that previous work provides 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 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 the EH code. We developed a tool, called XaviEH, which employs both coverage and mutation analysis as proxies for the effectiveness of EH testing in a certain libraries. Our findings suggest that EH code is, in general, less covered than regular code (i.e., non-EH).

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 these test suites are 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 can detect most of the 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. 2021). 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 a software evolution scenario; (iii) exploring libraries 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.

References

Ahmed I, Gopinath R, Brindescu C, Groce A, Jensen C (2016) Can testedness be effectively measured? In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, ACM, New York, NY, USA, FSE 2016, pp 547–558. https://doi.org/10.1145/2950290.2950324

Ammann P, Offutt J (2016) Introduction to software testing, 2nd edn. Cambridge University Press, Cambridge

Antinyan V, Derehag J, Sandberg A, Staron M (2018) Mythical unit test coverage. IEEE Softw 35(3):73–79. https://doi.org/10.1109/MS.2017.3281318

Barbosa EA, Garcia A, Barbos SDJ (2014) Categorizing faults in exception handling: A study of open source projects. In: 2014 Brazilian Symposium on Software Engineering (SBES), pp 11–20

Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. J Syst Softw 107:1–14

Benjamini Y, Yekutieli D (2001) The control of the false discovery rate in multiple testing under dependency. The Annals of Statistics 29(4):1165–1188. http://www.jstor.org/stable/2674075

Bloch J (2008) Effective java (The Java Series), 2nd edn. Prentice Hall PTR, Upper Saddle River

Cacho N, Barbosa EA, Araujo J, Pranto F, Garcia A, Cesar T, Soares E, Cassio A, Filipe T, Garcia I (2014a) How does exception handling behavior evolve? an exploratory study in java and c# applications. In: 2014 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 31–40

Cacho N, César T, Filipe T, Soares E, Cassio A, Souza R, Garcia I, Barbosa EA, Garcia A (2014b) Trading robustness for maintainability: An empirical study of evolving c# programs. In: Proceedings of the 36th international conference on software engineering, ACM, New York, NY, USA, ICSE 2014, pp 584–595. https://doi.org/10.1145/2568225.2568308

Chang BM, Choi K (2016) A review on exception analysis. Inf Softw Technol 77(C):1–16

Chekam TT, Papadakis M, Le TraonY, Harman M (2017) An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp 597–608. https://doi.org/10.1109/ICSE.2017.61

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

Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol Bull 114(4):494–509

Coelho R, Almeida L, Gousios G, Deursen AV, Treude C (2017) Exception handling bug hazards in android. Empirical Softw Engg 22(3):1264–1304

Dalton F, Ribeiro M, Pinto G, Fernandes L, Gheyi R, Fonseca B (2020) Is exceptional behavior testing an exception? an empirical assessment using java automated tests. In: Proceedings of the Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, EASE '20, pp 170–179. https://doi.org/10.1145/3383219.3383237

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30. http://dl.acm.org/citation.cfm?id=1248547.1248548

Digkas G, Lungu M, Avgeriou P, Chatzigeorgiou A, Ampatzoglou A (2018) 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), pp 153–163. https://doi.org/10.1109/SANER.2018.8330205

Ebert F, Castor F, Serebrenik A (2015) An exploratory study on exception handling bugs in java programs. J Syst Softw 106(C):82–101

Eck M, Palomba F, Castelluccio M, Bacchelli A (2019) 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, ACM, New York, NY, USA, ESEC/FSE, pp 830–840. https://doi.org/10.1145/3338906.3338945

Filho JLM, Rocha L, Andrade R, Britto R (2017) Preventing erosion in exception handling design using static-architecture conformance checking. In: Proceedings of the 11th European Conference on Software Architecture, Springer International Publishing, Cham, ECSA '17, pp 67–83. https://doi.org/10.1007/978-3-319-65831-5_5

Friedman M (1940) A comparison of alternative tests of significance for the problem of m rankings. The Annals of Mathematical Statistics 11(1):86–92

Gallardo R, Hommel S, Kannan S, Gordon J, Zakhour SB (2014) The Java tutorial: a short course on the basics. Java Series, 6th edn. Addison-Wesley Professional, Boston

Garcia AF, Rubira CM, Romanovsky A, Xu J (2001) A comparative study of exception handling mechanisms for building dependable object-oriented software. J Syst Softw 59(2):197–222

Gligoric M, Groce A, Zhang C, Sharma R, Alipour MA, Marinov D (2013) Comparing non-adequate test suites using coverage criteria. In: Proceedings of the 2013 International symposium on software testing and analysis, association for computing machinery, New York, NY, USA, ISSTA 2013, pp 302–313. https://doi.org/10.1145/2483760.2483769

Goffi A, Gorla A, Ernst MD, Pezzè M (2016) Automatic generation of oracles for exceptional behaviors. In: Proceedings of the 25th International symposium on software testing and analysis, association for computing machinery, New York, NY, USA, ISSTA 2016, pp 213–224. https://doi.org/10.1145/2931037.2931061

Goodenough JB, Gerhart SL (1975) Toward a theory of test data selection. IEEE Trans Softw Eng (2)156–173

Gopinath R, Jensen C, Groce A (2014) Code coverage for suite evaluation by developers. In: Proceedings of the 36th International conference on software engineering, association for computing machinery, New York, NY, USA, ICSE 2014, pp 72–82. https://doi.org/10.1145/2568225.2568278

Gulati S, Sharma R (2017) Java unit testing with JUnit 5: Test Driven Development with JUnit 5, 1st edn. Apress, USA

Hilton M, Bell J, Marinov D (2018) A large-scale study of test coverage evolution. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ACM, New York, NY, USA, ASE 2018, pp 53–63. https://doi.org/10.1145/3238147.3238183

Hunt A, Thomas D (2003) Pragmatic Unit Testing in Java with JUnit. The pragmatic programmers

Inozemtseva L, Holmes R (2014a) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International conference on software engineering, association for computing machinery, New York, NY, USA, ICSE 2014, pp 435–445. https://doi.org/10.1145/2568225.2568271

Inozemtseva L, Holmes R (2014b) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International conference on software engineering, ACM, New York, NY, USA, ICSE 2014, pp 435–445. https://doi.org/10.1145/2568225.2568271

Ivanković M, Petrović G, Just R, Fraser G (2019) 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, ACM, New York, NY, USA, ESEC/FSE 2019, pp 955–963. https://doi.org/10.1145/3338906.3340459

Jenkov J (2013) Java Exception Handling. Jenkov Aps

Ji C, Chen Z, Xu B, Wang Z (2009) A new mutation analysis method for testing java exception handling. In: Computer software and applications conference, 2009. COMPSAC'09. 33rd Annual IEEE International, vol 2. IEEE, pp 556–561

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) 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, ACM, New York, NY, USA, FSE 2014, pp 654–665. https://doi.org/10.1145/2635868.2635929

Kechagia M, Spinellis D (2014) Undocumented and unchecked: Exceptions that spell trouble. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR 2014, pp 312–315

Koch S (2007) Software evolution in open source projects—a large-scale investigation. J Softw Maint Evol 19(6):361–382

Kochhar PS, Thung F, Lo D (2015) 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), pp 560–564. https://doi.org/10.1109/SANER.2015.7081877

Kochhar PS, Lo D, Lawall J, Nagappan N (2017) Code coverage and postrelease defects: A large-scale study on open source projects. IEEE Trans Reliab 66(4):1213–1228. https://doi.org/10.1109/TR.2017.2727062

Kumar K, Gupta P, Parjapat R (2011) New mutants generation for testing java programs. In: International conference on advances in communication, network, and computing. Springer, pp 290–294

Lima L, Rocha L, Bezerra C, Paixao M (2021) Replication package for the paper: "assessing exception handling testing practices in open-source software systems. https://zenodo.org/record/4732368, Accessed: May 01, 2021

Luo Q, Hariri F, Eloussi L, Marinov D (2014) An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 643–653

Marinescu C (2011) Are the classes that use exceptions defect prone? In: Proceedings of the 12th International workshop on principles of software evolution and the 7th annual ERCIM workshop on software evolution. ACM, pp 56–60

Marinescu C (2013) Should we beware the exceptions? an empirical study on the eclipse project. In: 2013 15th International symposium on symbolic and numeric algorithms for scientific computing (SYNASC). IEEE, pp 250–257

Martins AL, Hanazumi S, d Melo ACV (2014) Testing java exceptions: An instrumentation technique. In: 2014 IEEE 38th international computer software and applications conference workshops, pp 626–631. https://doi.org/10.1109/COMPSACW.2014.105

McMinn P (2004) Search-based software test data generation: a survey. Software Testing, Verification and reliability 14(2):105–156

Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013. ACM Press, New York, p 466

Oliveira J, Borges D, Silva T, Cacho N, Castor F (2018) Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. J Syst Softw 136(C):1–18. https://doi.org/10.1016/j.jss.2017.10.032

Osman H, Chiş A, Corrodi C, Ghafari M, Nierstrasz O (2017) Exception evolution in long-lived java systems. In: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR '17, pp 302–311

de PáduaGB, Shang W (2017a) Revisiting exception handling practices with exception flow analysis. In: 2017 IEEE 17th International working conference on source code analysis and manipulation (SCAM), pp 11–20. https://doi.org/10.1109/SCAM.2017.16

de Pádua GB, Shang W (2017b) Studying the prevalence of exception handling anti-patterns. In: Proceedings of the 25th International conference on program comprehension. IEEE Press, Piscataway, NJ, USA, ICPC'17, pp 328–331

de Pádua GB, Shang W (2018) Studying the relationship between exception handling practices and post-release defects. In: Proceedings of the 15th International conference on mining software repositories. ACM, New York, NY, USA, MSR '18, pp 564–575. https://doi.org/10.1145/3196398.3196435

Paixao M, Krinke J, Han D, Ragkhitwetsagul C, Harman M (2017) 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

Papadakis M, Shin D, Yoo S, Bae DH (2018) 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. ACM, New York, NY, USA, ICSE '18, pp 537–548. https://doi.org/10.1145/3180155.3180183

Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M (2019) Chapter six - mutation testing advances: An analysis and survey. Advances in Computers. Elsevier 112:275–378

Pawlak R, Monperrus M, Petitprez N, Noguera C, Seinturier L (2016) Spoon: A library for implementing analyses and transformations of java source code. Softw Pract Exper 46(9):1155–1179. https://doi.org/10.1002/spe.2346

Rashkovits R, Lavy I (2012) Students' misconceptions of java exceptions. In: Rahman H, Mesquita A, Ramos I, Pernici B (eds) Proceedings of the 7th Mediterranean conference on information systems. Springer, Berlin, MCIS '12, pp 1–21. https://doi.org/10.1007/978-3-642-33244-9_1

Reales P, Polo M, Fernández-Alemán JL, Toval A, Piattini M (2014) Mutation testing. IEEE Software 31(3):30–35. https://doi.org/10.1109/MS.2014.68

Romano D, Di Penta M, Antoniol G (2011) An approach for search based testing of null pointer exceptions. In: 2011 Fourth IEEE international conference on software testing, verification and validation. IEEE, pp 160–169
Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR (2018) Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 15th international conference on mining software repositories, association for computing machinery, New York, NY, USA, MSR '18, pp 10–13. https://doi.org/10.1145/3196398.3196473
Sawadpong P, Allen EB (2016) Software defect prediction using exception handling call graphs: A case study. In: 2016 IEEE 17th International symposium on high assurance systems engineering (HASE). IEEE, pp 55–62
Schwartz A, Puckett D, Meng Y, Gay G (2018) Investigating faults missed by test suites achieving high code coverage. J Syst Softw 144:106–120. https://doi.org/10.1016/j.jss.2018.06.024
Shah H, Harrold MJ (2009) Exception handling negligence due to intra-individual goal conflicts. In: Proceedings of the 2009 ICSE workshop on cooperative and human aspects on software engineering. IEEE Computer Society, Washington, DC, USA, CHASE '09, pp 80–83. https://doi.org/10.1109/CHASE.2009.5071417
Shah H, Görg C, Harrold MJ (2008) Why do developers neglect exception handling? In: Proceedings of the 4th International Workshop on Exception Handling, ACM, New York, NY, USA, WEH '08, pp 62–68. https://doi.org/10.1145/1454268.1454277
Shah H, Gorg C, Harrold MJ (2010) Understanding exception handling: Viewpoints of novices and experts. IEEE Trans Softw Eng 36(2):150–161
Shahrokni A, Feldt R (2013) A systematic review of software robustness. Inf Softw Technol 55(1):1–17
Shi L, Zhong H, Xie T, Li M (2011) 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. Springer, Berlin, Heidelberg, FASE'11/ETAPS'11, pp 416–431
Sinha S, Harrold MJ (2000) Analysis and testing of programs with exception handling constructs. IEEE Trans Softw Eng 26(9):849–871. https://doi.org/10.1109/32.877846
Turner AJ, White DR, Drake JH (2016) Multi-objective regression test suite minimisation for mockito. In: Sarro F, Deb K (eds) Search based software engineering. Springer International Publishing, Cham, pp 244–249
Vieira R, da Silva A, Rocha L, Gomes JaP (2019) 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, Association for Computing Machinery, New York, NY, USA, PROMISE'19, pp 80–89. https://doi.org/10.1145/3345629.3345639
Wirfs-Brock RJ (2006) Toward exception-handling best practices and patterns. IEEE Softw 23(5):11–13. https://doi.org/10.1109/MS.2006.144
Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A (2012) Experimentation in software engineering. Springer Publishing Company Incorporated
Yang Y, Zhou Y, Sun H, Su Z, Zuo Z, Xu L, Xu B (2019) Hunting for bugs in code coverage tools via randomized differential testing. In: Proceedings of the 41st international conference on software engineering. IEEE Press, Piscataway, NJ, USA, ICSE '19, pp 488–499. https://doi.org/10.1109/ICSE.2019.00061
Zhai H, Casalnuovo C, Devanbu P (2019) Test coverage in python programs. In: Proceedings of the 16th international conference on mining software repositories. IEEE Press, Piscataway, NJ, USA, MSR '19, pp 116–120. https://doi.org/10.1109/MSR.2019.00027
Zhang P, Elbaum S (2014) Amplifying tests to validate exception handling code: An extended study in the mobile application domain. ACM Trans Softw Eng Methodol 23(4):32:1–32:28. https://doi.org/10.1145/2652483
Zhong H, Mei H (2019) An empirical study on api usages. IEEE Trans Softw Eng 45(4):319–334. https://doi.org/10.1109/TSE.2017.2782280
Zhu H, Hall PAV, May JHR (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427. https://doi.org/10.1145/267580.267590

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Luan P. Lima[1] · Lincoln S. Rocha[1] · Carla I. M. Bezerra[2] · Matheus Paixao[3]

Luan P. Lima
luanpereira@great.ufc.br

Carla I. M. Bezerra
carlailane@ufc.br

Matheus Paixao
matheus.paixao@uece.br

[1] Federal University of Ceará, Fortaleza CE, Brazil

[2] Federal University of Ceará, Quixadá CE, Brazil

[3] State University of Ceará, Fortaleza CE, Brazil